Name: James Lee
ID: 6950219
GitHub link: https://github.coventry.ac.uk/leej64/210CT-Course-Work

All of the advanced tasks completed and none of the basic tasks

Task 1 Part 1:

main.py

```
#!/usr/bin/python3
"""main.py: contains the dirver code the show the working functions from
'factorials.py'
"""

from factorials import test_divides


def main():
    """main: Driver code to show that factorial and divides functions work
    """
    test_values = [(6, 9), (20, 10000), (6, 27), (20, 1000000)]
    for num_a, num_b in test_values:
        print(test_divides(num_a, num_b)[0])


main()
```

factorials.py

```
#!/usr/bin/python3
"""factorials.py: contains the functions to calculate the factorial of a number
and to test dividing equally with a factorial number
"""


def factorial(num):
    """factorial

    :param num: Number which you would like the factorial of
    """
    assert isinstance(num, int)
    if num == 0:
        return 1
    return num * factorial(num - 1)


def test_divides(num_a, num_b):
    """test_divides
    :param num_a: Number we calculate the factorial of and then is tested if
    divides by 'b' equally
    :param num_b: 'a!' is divided by this number
    """
    assert isinstance(num_a, int)
    assert isinstance(num_b, int)

    if factorial(num_a) % num_b:
        return("{0} does not divide by {1}!".format(num_b, num_a), False)
    return("{0} divides by {1}!".format(num_b, num_a), True)
```

unit_test.py

```
#!/usr/bin/python3
"""unit test.py: The class to run tests on functions from 'factorials.py'
"""

import unittest
from factorials import test_divides


class UnitTest(unittest.TestCase):
    """UnitTest"""
    def test_correct(self):
        """test_correct: Test known correct values taken from the labsheet
        """
        known_correct_values = [(6, 9), (20, 10000)]
        for num_a, num_b in known_correct_values:
            self.assertTrue(test_divides(num_a, num_b)[1])

    def test_false(self):
        """test_false: Test known wrong values taken from the labsheet
        """
        known_wrong_values = [(6, 27), (20, 1000000)]
        for num_a, num_b in known_wrong_values:
            self.assertFalse(test_divides(num_a, num_b)[1])


if __name__ == '__main__':
    unittest.main()
```

pseudo_code.txt

```
FACTORIAL(n)
    IF n = 0
        RETURN 1
    ELSE
        RETURN n * FACTORIAL(n - 1)
```

```
TEST-IF-DEVIDES(a, b)
    IF a! MOD b = 0
        RETURN true
    ELSE
        RETURN false
```

Task 1 Part 2

lorry.py

```
#!/usr/bin/python3
"""lorry.py: Contains the class for the lorry object which is used
to calculate the most expensive load
"""



class Lorry:
    """Lorry: Lorry class representing a lorry which is used to calculate the
    most expensive load
    """
    def __init__(self, maxLoad):
        """__init__

        :param maxLoad: Maximum load capacity of the lorry (r
        """
        assert isinstance(maxLoad, int)
        self.max_load = maxLoad
        self.load_composition = 0
        self.cargo = {}

    def pickup_delivery(self, materials):
        """pickup_delivery: fills the lorrys cargo with the most profitable load

        :param materials: List of availble materials
        """
        assert isinstance(materials, list)
        while self._current_weight() < self.max_load:
            material = self.most_expensive_material(materials)
            if material.name not in self.cargo:
                self.cargo[material.name] = 0
            self.cargo[material.name] += 1
            self.load_composition += material.price_per_kilo
            material.decriment()

    def _current_weight(self):
        return sum(self.cargo.values())

    def __str__(self):
        info = 'Load composition value = {0}\n'.format(self.load_composition)
        for key in self.cargo:
            info = info + '{0}kg of {1} and '.format(self.cargo[key], key)
        info = info[:-4]
        return info

    @classmethod
    def most_expensive_material(cls, materials):
        """most_expensive_material

        :param materials: List of availble materials
        :return material: The most expensive material in the list
        """
        assert isinstance(materials, list)
        cost = 0
        for material in materials:
            if material > cost and material.quantity > 0:
                cost = material.price_per_kilo
                most_expensive = material
        return most_expensive
```

main.py

```
#!/usr/bin/python3
"""main.py: Contains the driver code the show the working functions from
'lorry.py' and 'material.py'
"""

from lorry import Lorry
from material import Material


def main():
    """main: Driver code to show working implimentation of labsheet question
    """
    gold = Material('Gold', 4, 100)
    copper = Material('Copper', 7, 65)
    plastic = Material('Plastic', 15, 50)

    materials = [gold, plastic, copper]
    lorry1 = Lorry(10)
    lorry1.pickup_delivery(materials)
    print(lorry1)


main()
```

material.py

```
#!/usr/bin/python3
"""material.py: Class representing a material and is used in conjunction with
'lorry' to figure out the best load composition
"""


class Material:
    """material: Material class representing a possible material which can be
    loaded into a lorry """
    def __init__(self, name, quantity, price_per_kilo):
        assert isinstance(name, str)
        assert isinstance(quantity, int)
        assert isinstance(price_per_kilo, int)

        self.name = name
        self.price_per_kilo = price_per_kilo
        self.quantity = quantity

    def decriment(self):
        self.quantity -= 1

    def __gt__(self, other):
        if isinstance(other, int):
            return self.price_per_kilo > other
        elif isinstance(other, Material):
            return self.price_per_kilo > other.price_per_kilo
        return False

    def __str__(self):
        return 'Name: {0}\nPPK: {1}\nQuantity: {2}\n'.format(self.name, self.price_per_kilo,
                                                             self.quantity)
```

unit_test.py

```
#!/usr/bin/python3
"""unit_test.py: Unit testing for the Lorry class
"""

import unittest
from material import Material
from lorry import Lorry


class UnitTest(unittest.TestCase):
    """UnitTest"""
    def test_labsheet(self):
        """test_labsheet: Test labsheet values
        """
        known_correct_values = {"Gold": 4, "Copper": 6}
        gold = Material('Gold', 4, 100)
        copper = Material('Copper', 7, 65)
        plastic = Material('Plastic', 15, 50)

        materials = [gold, plastic, copper]
        lorry1 = Lorry(10)
        lorry1.pickup_delivery(materials)

        self.assertEqual(lorry1.cargo, known_correct_values)
        self.assertEqual(lorry1.load_composition, 790)

    def test_extra(self):
        """test_extra: Test extra values to make sure the code should work with any material
        objects
        """
        known_correct_values = {"Ruby": 2, "Copper": 7, "Diamond": 1, "Plastic": 1, "Gold": 4}

        gold = Material('Gold', 4, 100)
        copper = Material('Copper', 7, 65)
        plastic = Material('Plastic', 15, 50)
        diamond = Material('Diamond', 1, 1000)
        ruby = Material('Ruby', 2, 500)

        materials = [gold, plastic, copper, diamond, ruby]

        lorry1 = Lorry(15)
        lorry1.pickup_delivery(materials)

        self.assertEqual(lorry1.cargo, known_correct_values)
        self.assertEqual(lorry1.load_composition, 2905)


if __name__ == '__main__':
    unittest.main()
```

Task 2

main.py

```
#!/usr/bin/python3
"""main.py: Holds the boilerplate code to show that this code solves
the 8 queens problem
"""

from queen import Queen


def main():
    """main: Code to show the 8 queens problem being solved
    """
    for i in range(8):
        solver = Queen(8)
        print("Solution {0} - {1}".format(i + 1, solver.place_queen(i)))


main()
```

pseudo_code.txt

```
solve(pos=0)
  boardState = []
  IF LEN(boardState)
      FOR i = pos; pos < 8
          IF safe_placement(LEN(boardState))
              boardState APPEND i
              RETURN solve()
      lastQueenRow = boardState.pop()
      return solve(lastQueenRow + 1)
```

queen.py

```
#!/usr/bin/python3
"""queen.py: Holds the Queen class which is a solver for the 8 queens problem
"""



class Queen():
    """Queen: Solver for the eight queen problem
    """
    def __init__(self, bs):
        """__init__

        :param bs: int representing the size of the board

        board State [1, 3, 0, 2] = [ - Q - - ]
                                   [ - - - Q ]
                                   [ Q - - - ]
                                   [ - - Q - ]
        """
        assert isinstance(bs, int) and bs <= 8
        self.board_size = bs
        self.board_state = []

    def is_safe(self, pos_x, pos_y):
        """is_safe: Checks if its safe to place a queen at position 'x, y'

        :complexity: O(n) where is is the size of the board
        :param pos_x: int representing column
        :param pos_y: int representing row
        """
        assert isinstance(pos_x, int)
        assert isinstance(pos_y, int)
        for col in range(len(self.board_state)):
            row = self.board_state[col]
            if pos_x == col or pos_y == row:
                return False
            if pos_x + pos_y == col + row or pos_x - pos_y == col - row:
                return False
        return True

    def place_queen(self, pos=0):
        """place_queen: Recursively place a queen until you can longer place a queen backtrack
        until you can place another queen

        :complexity: O(N!)
        :param pos: Int representing position on the board
        """
        assert isinstance(pos, int) and pos <= self.board_size
        if len(self.board_state) == self.board_size:
            return self.board_state
        for col in range(pos, self.board_size):
            if self.is_safe(len(self.board_state), col):
                self.board_state.append(col)
                return self.place_queen()
        last_queen_row = self.board_state.pop()
        return self.place_queen(last_queen_row + 1)
```

unit_test.py

```
#!/usr/bin/python3
"""unit_test.py: Unit testing for the Queen class
"""

import unittest
from queen import Queen


class UnitTest(unittest.TestCase):
    """UnitTest: Unit testing for the 8 queens problem solver
    """
    def test_correct(self):
        """test_correct: Test generated values from solver and make sure that they
        are in the 'known_correct_values' list"""
        known_correct_values = [[0, 4, 7, 5, 2, 6, 1, 3],
                                [1, 3, 5, 7, 2, 0, 6, 4],
                                [2, 0, 6, 4, 7, 1, 3, 5],
                                [3, 0, 4, 7, 1, 6, 2, 5],
                                [4, 0, 3, 5, 7, 1, 6, 2],
                                [5, 0, 4, 1, 7, 2, 6, 3],
                                [6, 0, 2, 7, 5, 3, 1, 4],
                                [7, 1, 3, 0, 6, 4, 2, 5]]

        for i in range(8):
            solver = Queen(8)
            self.assertTrue(solver.place_queen(i) in known_correct_values)

    def test_is_safe(self):
        """test_is_safe: Test to make sure that is_safe function works
        """
        solver = Queen(8)
        solver.board_state = [0]
        self.assertFalse(solver.is_safe(0, 1))
        self.assertFalse(solver.is_safe(1, 0))
        self.assertFalse(solver.is_safe(1, 1))
        self.assertTrue(solver.is_safe(2, 1))
        self.assertTrue(solver.is_safe(1, 2))


if __name__ == '__main__':
    unittest.main()
```

Task 3

cube.py

```
#!/usr/bin/python3
"""cube.py: Class file which contains the Cube class this class is used in conjunction with
stacking.py to stack cubes according to a set of rules
"""



class Cube:
    """cube: represents a cube but could easy be replaced by a dictionary, which would also be
    much faster"""
    def __init__(self, color, edge_length):
        assert isinstance(color, str)
        assert isinstance(edge_length, int)
        self.color = color
        self.edge_length = edge_length
```

main.py

```
#!/usr/bin/python3
"""main.py: File contains boilerplate code to test stacking.py
"""
```

```
from cube import Cube
from stacking import stack_cubes


def main():
    """main: Driver function to make sure code is running correctly """
    cube1 = Cube('red', 5)
    cube2 = Cube('red', 6)
    cube3 = Cube('blue', 5)
    cube_list = [cube1, cube2, cube3]
    print(stack_cubes(cube_list))


main()
```

stacking.py

```
#!/usr/bin/python3
"""stacking.py: Contains the functions to stack cubes according to rules
"""


def calc_height(stacked_list):
    """calc_height

    :param stacked_list: List of cubes
    :returns height: height of stacked cubes
    """
    assert isinstance(stacked_list, list)
    return 'The maximum tower height is {0}'.format(sum([i.edge_length for i in stacked_list]))


def widest_cube(cube_list, stacked_list):
    """widest_cube

    :param cube_list: List of cube objects
    :param stacked_list: List of current stacked cube objects
    """
    assert isinstance(cube_list, list)
    assert isinstance(stacked_list, list)
    current_widest_cube = None
    widest_edge = 0
    color = None
    if stacked_list:
        color = stacked_list[-1].color
    for cube in cube_list:
        if cube.edge_length > widest_edge and cube.color != color and cube not in stacked_list:
            widest_edge = cube.edge_length
            current_widest_cube = cube
    return current_widest_cube


def stack_cubes(cube_list):
    """stack_cubes

    :param cube_list: List of availble cube objects
    """
    assert isinstance(cube_list, list)
    stacked_list = []
    stacked_list.append(widest_cube(cube_list, stacked_list))
    while len(stacked_list) != len(cube_list):
        stacked_list.append(widest_cube(cube_list, stacked_list))
        if None in stacked_list:
            raise ValueError('You cannot stack these cubes according to the rules')
    return calc_height(stacked_list)
```

unit_test.py

```
#!/usr/bin/python3
"""unit_test: File which contains the unit testing to make sure that
functions from stacking.py work as intended
"""

import unittest
from cube import Cube
from stacking import calc_height, stack_cubes, widest_cube


class UnitTest(unittest.TestCase):
    """UnitTest"""
    def test_calc_height(self):
        """test_calc_height: Testing calculate_height function"""
        cube1 = Cube('red', 6)
        cube2 = Cube('blue', 5)
        stacked_list = [cube1, cube2]
        self.assertEqual(calc_height(stacked_list), 'The maximum tower height is 11')

    def test_failure(self):
        """test_failure: Make sure a ValueError is raised if you cannot stack the cubes"""
        cube1 = Cube('red', 5)
        cube2 = Cube('red', 5)
        cube_list = [cube1, cube2]
        with self.assertRaises(ValueError):
            stack_cubes(cube_list)

    def test_widest_cube(self):
        """test_widest_cube: Test function to find the next widest cube
        """
        cube1 = Cube('red', 5)
        cube2 = Cube('blue', 3)
        cube3 = Cube('red', 5)
        cube4 = Cube('green', 6)
        cube5 = Cube('purple', 7)
        cube6 = Cube('red', 2)
        cube_list = [cube1, cube2, cube3, cube4, cube5, cube6]
        stacked_list = [cube5]
        self.assertEqual(widest_cube(cube_list, stacked_list), cube4)


if __name__ == '__main__':
    unittest.main()
```

Task 4 Part 1

main.py

```
#!/usr/bin/python3
"""main.py: Boilerplate code to test set finding functions
"""

from set_finding import find_largest_set, create_array


def main():
    """main: Main driver code to ask user for input then use quick_sort
    """
    array = create_array(8, 8)
    number_set = find_largest_set(array)

    for index, number in enumerate(number_set):
        if not number:
            print('There are no sets of numbers in the matrix')
        print('{0}. Number/Color = {1}\n   Set = {2}\n'.format(index + 1,
                                                               array[number[0][0]][number[0][1]],
                                                               number))


main()
```

set_finding.py

```
#!/usr/bin/python3
"""set_finding.py: Collection of functions which allow you to find sets
of numbers in a matrix of integers
"""
```

```python
import numpy as np


def check_if_set(array, pos_x, pos_y):
    """check_if_set: Recursive method to check if the position (x, y) is included in a set.

    :param array: Matrix in which we are checking for sets
    :param pos_x: Int representing x coord
    :param pos_y: Int representing y coord
    :returns list: List of visited coords
    """
    assert isinstance(array, (np.ndarray, list))
    assert isinstance(pos_x, int)
    assert isinstance(pos_y, int)

    def _check_if_set(array, pos_x, pos_y, visited):
        neighbours = check_neighbours(array, pos_x, pos_y)
        for pos_i, pos_j in neighbours:
            if (pos_i, pos_j) not in visited:
                visited.append((pos_i, pos_j))
                _check_if_set(array, pos_i, pos_j, visited)
        return visited

    return _check_if_set(array, pos_x, pos_y, [])


def check_neighbours(array, pos_x, pos_y):
    """check_neighbours:  Returns a list of direct neighbors with the same color as pos (x, y)

    :param array: Matrix
    :param pos_x: Int representing x coord
    :param pos_y: Int representing y coord
    """
    assert isinstance(array, (np.ndarray, list))
    assert isinstance(pos_x, int)
    assert isinstance(pos_y, int)
    neighbours = []
    if not (pos_x - 1) < 0:
        if array[pos_x - 1][pos_y] == array[pos_x][pos_y]:
            neighbours.append((pos_x - 1, pos_y))

    if not (pos_x + 1) > array.shape[0] - 1:
        if array[pos_x + 1][pos_y] == array[pos_x][pos_y]:
            neighbours.append((pos_x + 1, pos_y))

    if not (pos_y - 1) < 0:
        if array[pos_x][pos_y - 1] == array[pos_x][pos_y]:
            neighbours.append((pos_x, pos_y - 1))
    if not (pos_y + 1) > array.shape[1] - 1:
        if array[pos_x][pos_y + 1] == array[pos_x][pos_y]:
            neighbours.append((pos_x, pos_y + 1))

    return neighbours


def get_all_sets(array):
    """get_all_sets:  Method to get all sets of colors in the matrix.
    This includes empty sets which will are removed in "find_largest_set"

    :param array: Matrix which we are searching for sets of numbers
    """
    assert isinstance(array, (np.ndarray, list))
    all_sets = []
    for i in range(array.shape[0]):
        for j in range(array.shape[1]):
            all_sets.append(check_if_set(array, i, j))
    return all_sets


def find_largest_set(array):
    """find_largest_set: Uses list from "get_all_sets" and finds the largest set of numbers
    next to each other in the matrix if there is multiple sets which are the largest
    they will all be returned.

    :param array: Matrix which we are finding the largest set in
    :returns  list: A list of lists containing tuples of x y coordinates
    """
    assert isinstance(array, (np.ndarray, list))
    sets = []
    current_largest_set = 0
    for lst in get_all_sets(array):
        if len(lst) > current_largest_set:
            sets = []
            lst.sort()
            if lst not in sets:
                sets.append(lst)
                current_largest_set = len(lst)
        elif len(lst) == current_largest_set:
            lst.sort()
            if lst not in sets:
                sets.append(lst)
    return sets


def create_array(size_x, size_y):
    """create_array: Helper function to create the array and print it. Needed due to the fact that
    the array is full or pseudo random data and we need to be able to see the array
    to know if the output is correct

    :param n: int width of array to be created
    :param m: int height of array to be created
    """
    assert isinstance(size_x, int)
    assert isinstance(size_y, int)
    array = np.random.random_integers(1, 9, size=(size_x, size_y))
    print("{0}\n".format(array))
    return array
```

unit_test.py

```
#!/usr/bin/python3
"""unit_test.py: File contains all the unit testing to make sure that the functions
in 'set_finding.py' are correct
"""

import unittest
import numpy as np
from set_finding import check_neighbours, check_if_set, find_largest_set


class UnitTest(unittest.TestCase):
    """UnitTest"""
    def test_check_neighbours(self):
        """test_check_neighbours: Test case the neightbour checking function
        """
        # left
        array = np.zeros((4, 4), int)
        array[1][1] = 1
        array[1][0] = 1

        self.assertEqual(check_neighbours(array, 1, 1), [(1, 0)])

        # right
        array = np.zeros((4, 4), int)
        array[1][1] = 1
        array[1][2] = 1

        self.assertEqual(check_neighbours(array, 1, 1), [(1, 2)])

        # above
        array = np.zeros((4, 4), int)
        array[1][1] = 1
        array[0][1] = 1

        self.assertEqual(check_neighbours(array, 1, 1), [(0, 1)])
        # below
        array = np.zeros((4, 4), int)
        array[1][1] = 1
        array[2][1] = 1

        self.assertEqual(check_neighbours(array, 1, 1), [(2, 1)])

    def test_check_if_set(self):
        """test_check_if_set: Testing check_if_set function is working correctly
        """
        array = np.zeros((4, 4), int)
        array[1][1] = 1
        array[0][1] = 1
        array[2][1] = 1
        array[1][2] = 1
        array[1][0] = 1

        returned_value = check_if_set(array, 1, 1)
        returned_value.sort()

        correct = [(0, 1), (1, 1), (2, 1), (1, 0), (1, 2)]
        correct.sort()

        self.assertEqual(returned_value, correct)

    def test_find_largest_set(self):
        """test_find_largest_set: Test main function which should return any sets of numbers in
        a matrix
        """
        array = np.zeros((4, 4), int)

        array[1][1] = 1
        array[0][1] = 1
        array[2][1] = 1
        array[1][2] = 1
        array[1][0] = 1

        self.assertEqual(find_largest_set(array), [[(0, 2), (0, 3), (1, 3), (2, 0), (2, 2),
                                                    (2, 3), (3, 0), (3, 1), (3, 2), (3, 3)]])

        array = np.zeros((4, 4), int)

        for i in range(array.shape[0]):
            for j in range(array.shape[1]):
                array[i][j] = 1

        self.assertEqual(find_largest_set(array), [[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0),
                                                    (1, 1), (1, 2), (1, 3), (2, 0), (2, 1),
                                                    (2, 2), (2, 3), (3, 0), (3, 1), (3, 2),
                                                    (3, 3)]])


if __name__ == '__main__':
    unittest.main()
```

Task 4 Part 2

main.py

```
#!/usr/bin/python3
"""main.py: Code to use quick sort to get an input from the user and return that
indexes value from the sorted list
"""

import random
from sorting import quick_sort


def ordinal(num):
    """ordinal: Generate an ordinal number representation of 'num'

    :param num: Integer which you want the ordinal representation of
    """
    if num >= 10 and num <= 20:
        suffix = 'th'
    else:
        suffix = {1: 'st', 2: 'nd', 3: 'rd'}.get(num % 10, 'th')
    return str(num) + suffix


def main():
    """ generate an array of length '10' sort it and as the user which element they would like """

    sorted_array = quick_sort([random.randint(1, 1000) for i in range(10)])

    while True:
        try:
            element = int(input('Which element do you want to find? '))
            break
        except ValueError:
            print("Please enter a integer between 1 and {0}".format(len(sorted_array)))

    print("{0}".format(sorted_array))

    try:
        if element > len(sorted_array) // 2:
            print('The {0} largest element is {1}'.format(ordinal(element),
                                                          sorted_array[element - 1]))
        else:
            print('The {0} smallest element is {1}'.format(ordinal(element),
                                                          sorted_array[element - 1]))
    except IndexError:
        raise IndexError('Index is not in list')


main()
```

sorting.py

```
#!/usr/bin/python3
"""sorting.py: Simple implimentation of the quick sort algorithm
"""


def quick_sort(array):
    """quick_sort: Use the quick sort algorithm to sort 'array'

    :param array: List of integer, should work with strings
    :output array: Sorted version of 'array'
    """
    assert isinstance(array, list)
    less_list, greater_list, equal_list = [], [], []
    if len(array) <= 1:
        return array
    pivot_point = array[len(array) // 2]
    for item in array:
        if item == pivot_point:
            equal_list.append(item)
        elif item < pivot_point:
            less_list.append(item)
        else:
            greater_list.append(item)
    return quick_sort(less_list) + equal_list + quick_sort(greater_list)
```

unit_test.py

```
#!/usr/bin/python3
"""unit_test.py: Unit testing to make sure that quick sort has been implimented properly
"""

import unittest
from sorting import quick_sort


class UnitTest(unittest.TestCase):
    """UnitTest"""
    def test_quick__sort(self):
        """test_quick__sort: Make sure that i have implimented quick_sort properly
        """
        unsorted_list = [1, 5, 2, 6, 8, 5, 234, 5645, 234, 6, 4, 756, 234, 2, 3, 4, 656, 7, 234]
        self.assertEqual(quick_sort(unsorted_list), [1, 2, 2, 3, 4, 4, 5, 5, 6, 6, 7, 8, 234, 234,
                                                     234, 234, 656, 756, 5645])

        unsorted_list = [5, 3, 2, 72, 5, 7, 23]
        self.assertEqual(quick_sort(unsorted_list), [2, 3, 5, 5, 7, 23, 72])

        unsorted_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
        self.assertEqual(quick_sort(unsorted_list), [1, 2, 3, 4, 5, 6, 7, 8, 9])


if __name__ == '__main__':
    unittest.main()
```

Task 5 Part 1

diagonals.py

```
#!/usr/bin/python3
"""diagonals.py: File contains all the functions to get the diagonals from a matrix
"""


import numpy as np


def get_diagonal(array, offset=0):
    """get_diagonal: Method to get the main diagonal from matrix

    :param array: Matrix of integers
    :param offset: Integer which allows you to offset which diagonal you want to get e.g getting
    the diagonal above the main one
    """
    assert isinstance(array, (np.ndarray, list))
    assert offset <= len(array)
    assert offset >= -len(array)
    lst = []
    for index in range(len(array)):
        try:
            if offset > 0:
                lst.append(array[index + offset][index])
            else:
                lst.append(array[index][index + abs(offset)])
        except IndexError:
            pass
    return lst
```

```
def get_all_diagonals(array, size):
    """get_all_diagonals: Gets all the diagonals in the matrix

    :param array: Matrix of intergers
    :param size: Integer representing the size wanted
    """
    assert isinstance(array, (np.ndarray, list))
    assert isinstance(size, int)
    diagonals = []
    for i in range(-(len(array) - 1), (len(array))):
        diagonal = get_diagonal(array, i)
        if len(diagonal) >= size:
            diagonals.append(diagonal)
    return diagonals


def smallest_sum_in_array(array, size):
    """smallest_sum_in_array: gets the smallest sum of m elements in the array

    :param array: List of itegers
    :param size: int representing the amount of elements
    """
    assert isinstance(array, list)
    assert isinstance(size, int)
    array.sort()
    return sum(array[:size])
```

main.py

```
#!/usr/bin/python3
"""main.py: File contains the function to generate a random matrix and find the smallest
sum of integers in the diagonals
"""

import numpy as np
from diagonals import get_all_diagonals, smallest_sum_in_array


def main():
    """main"""
    array_size = 8
    diagonal_size = 4
    assert array_size > diagonal_size

    array = np.random.random_integers(1, 9, size=(array_size, array_size))
    diagonals = get_all_diagonals(array, diagonal_size)

    for index, diagonal in enumerate(diagonals):
        diagonals[index] = smallest_sum_in_array(diagonal, diagonal_size)

    print(array)
    print('Answer: {0}'.format(min(diagonals)))


main()
```

unit_test.py

```
#!/usr/bin/python3
"""unit_test.py: Testing to make sure that the functions from
diagonals.py is working correctly
"""

import unittest
from diagonals import get_all_diagonals, smallest_sum_in_array, get_diagonal


class UnitTest(unittest.TestCase):
    """UnitTest"""
    def test_get_diagonal(self):
        """test_get_diagonal: Make sure test_get_diagonal is working correctly
        """
        array = [[3, 1, 5, 6, 9], [2, 4, 1, 9, 7], [3, 5, 2, 8, 10], [4, 2, 1, 6, 8],
                 [1, 4, 7, 9, 1]]
        self.assertEqual(get_diagonal(array), [3, 4, 2, 6, 1])

    def test_sum_in_array(self):
        """test_sum_in_array: Make sure sum_in_array function is working correctly
        """
        array = [1, 123, 312, 3, 223, 1, 323, 4, 123, 1, 23, 1]
        self.assertEqual(smallest_sum_in_array(array, 4), 4)

    def test_labsheet(self):
        """test_labsheet: Test values given on the labsheet
        """
        array = [[3, 1, 5, 6, 9], [2, 4, 1, 9, 7], [3, 5, 2, 8, 10], [4, 2, 1, 6, 8],
                 [1, 4, 7, 9, 1]]

        diagonals = get_all_diagonals(array, 4)

        for i in range(len(diagonals)):
            diagonals[i] = smallest_sum_in_array(diagonals[i], 4)

        diagonals = min(diagonals)

        self.assertEqual(diagonals, 10)
```

```
if __name__ == '__main__':
    unittest.main()
```

Task 5 Part 2

main.py

```
#!/usr/bin/python3
"""main.py: Read a file and create linked lists of word size containing no repeating words
"""

import string
from linked_list import LinkedList
from node import Node


def main():
    """main: Opens up a text file and creates linked lists of words
    """
    with open('paragraph.txt') as file:
        linked_lists = {}
        words = file.read().split()
        for word in words:
            word = word.translate(str.maketrans('', '', string.punctuation))
            word = word.lower()

            new_node = Node(word)

            if len(word) not in linked_lists:
                linked_lists[len(word)] = LinkedList()
                linked_lists[len(word)].append(new_node)
            else:
                if not linked_lists[len(word)].is_in(word):
                    linked_lists[len(word)].append(new_node)

        for i in linked_lists:
            linked_lists[i].sort()
            print('Words of length {0}: {1}'.format(i, linked_lists[i]))


main()
```

linked_list.py

```
#!/usr/bin/python3
"""linked_list.py:
"""
```

```
from node import Node


class LinkedList:
    """LinkedList: Linked list class allowing the creation an manipulation of linked lists
    """
    def __init__(self):
        self.size = 0
        self.first_node = None
        self.last_node = None

    def push(self, new_node):
        """push: Put 'new_node' at the front of the linked list

        :param new_node: Node object
        """
        assert isinstance(new_node, Node)
        if self.first_node is None and self.last_node is None:
            self.first_node = new_node
            self.last_node = new_node
        else:
            new_node.next_node = self.first_node
            new_node.previous_node = None
            self.first_node.previous_node = new_node
            self.first_node = new_node
        self.size += 1

    def pop(self):
        """pop: Returns the node from the back of the linked list and removes it

        :returns node: Node from the back of the list
        """
        node = self.last_node
        self.remove(node)
        return node

    def is_in(self, word):
        """is_in: Checks if word is in list

        :param word: string
        """
        assert isinstance(word, str)
        current_node = self.first_node
        while current_node:
            if current_node.data == word:
                return True
            current_node = current_node.next_node
        return False
```

```
def append(self, new_node):
    """append: Add a new node to the end of the linked list

    :param new_node: Node which you want to append to the list
    """
    assert isinstance(new_node, Node)
    if self.first_node is None and self.last_node is None:
        self.first_node = new_node
        self.last_node = new_node
    else:
        new_node.next_node = None
        new_node.previous_node = self.last_node
        self.last_node.next_node = new_node
        self.last_node = new_node
    self.size += 1

def remove(self, node):
    """remove: Remove a node from the linked list

    :param node: Node to be removed from the linked list
    """
    assert isinstance(node, Node)
    if node == self.first_node:
        self.first_node = node.next_node
    elif node == self.last_node:
        self.last_node = node.previous_node
        self.last_node.next_node = None
    else:
        next_node = node.next_node
        previous_node = node.previous_node
        next_node.previous_node = previous_node
        previous_node.next_node = next_node
    self.size -= 1

def sort(self):
    """sort: Sorts the linked list
    """
    current_node = self.first_node
    is_sorted = False

    while not is_sorted:
        is_sorted = True
        while current_node:
            if current_node.next_node is not None and \
                    current_node.data > current_node.next_node.data:
                self._swap(current_node, current_node.next_node)
                is_sorted = False
            current_node = current_node.next_node
        current_node = self.first_node
@classmethod
def _swap(cls, node_a, node_b):
    """_swap: Swaps the data of two elements in the linked list

    :param node_a:
    :param node_b:
    """
    assert isinstance(node_a, Node)
    assert isinstance(node_b, Node)
    tmp_data = node_b.data
    node_b.data = node_a.data
    node_a.data = tmp_data

def __len__(self):
    return self.size

def __str__(self):
    contents = '['
    next_node = self.first_node
    while next_node:
        if next_node.next_node is None:
            contents += '{0}]'.format(str(next_node.data))
        else:
            contents += '{0}, '.format(str(next_node.data))
        next_node = next_node.next_node
    return contents
```

node.py

```
#!/usr/bin/python3
"""node.py: Linked list nodes
"""

class Node:
    """Node: node class for the linked lists
    """
    def __init__(self, data, previous_node=None, next_node=None):
        assert isinstance(data, str)
        assert isinstance(previous_node, Node) or previous_node is None
        assert isinstance(next_node, Node) or next_node is None
        self.data = data
        self.next_node = next_node
        self.previous_node = previous_node
```

unit_test.py

```
#!/usr/bin/python3
"""unit_test.py: Make sure that the LinkedList class is working correctly
```

```
"""

import unittest
from linked_list import LinkedList
from node import Node


class UnitTest(unittest.TestCase):
    """UnitTest"""
    def test_append(self):
        """test_append: Test appending to a linked list
        """
        node1 = Node('test')
        lst = LinkedList()
        lst.append(node1)
        self.assertTrue(len(lst) == 1)

    def test_pop(self):
        """test_pop: Test removing a node from the end of the linked list and returning it
        """
        node1 = Node('test')
        lst = LinkedList()
        lst.append(node1)
        lst.pop()
        self.assertTrue(len(lst) == 0)

    def test_swap(self):
        """test_swap: test swapping two nodes values
        """
        node1 = Node('test')
        node2 = Node('anotherTest')
        lst = LinkedList()
        lst.append(node1)
        lst.append(node2)
        lst._swap(node1, node2)

        current_node = lst.first_node
        while current_node:
            self.assertTrue(current_node.data == node1.data or node2.data)
            current_node = current_node.next_node

    def test_remove(self):
        """test_remove: Test removing a node by is reference
        """
        node1 = Node('test')
        node2 = Node('anotherTest')
        lst = LinkedList()
        lst.append(node1)
        lst.append(node2)
        lst.remove(node1)

        current_node = lst.first_node
        while current_node:
            self.assertTrue(current_node.data == node2.data)
            current_node = current_node.next_node


if __name__ == '__main__':
    unittest.main()
```

Task 6

address.py

```
#!/usr/bin/python3

class Address:
    def __init__(self, house_num, street_name):
        assert isinstance(house_num, int)
        assert isinstance(street_name, str)

        self.house_num = house_num
        self.street_name = street_name

    def __lt__(self, other):
        return self.house_num < other.house_num

    def __gt__(self, other):
        return self.house_num > other.house_num

    def __eq__(self, other):
        if str(other) == str(self):
            return True
        return False

    def __ne__(self, other):
        if str(other) == str(self):
            return False
        return True

    def __str__(self):
        return '{0} {1}'.format(self.house_num, self.street_name)
```

binary_tree.py

```
#!/usr/bin/python3
""" binary_tree.py: File which has the class for a simple binary tree
"""
```

```
from node import Node


class BinaryTree:
    """BinaryTree: Binary tree class which can store several different datatypes
    and is used in the simple database
    """
    def __init__(self):
        self.root = None
        self.type = None

    def insert(self, new_node):
        """insert: Insert a new node into the binary tree

        :param new_node: Node which is going to be put into the tree
        """
        assert isinstance(new_node, Node)
        if self.root is None:
            self.type = type(new_node.value)
            self.root = new_node
            return

        current_node = self.root
        while current_node:
            if new_node.value < current_node.value:
                if current_node.left_node is None:
                    new_node.parent = current_node
                    current_node.left_node = new_node
                    break
                else:
                    current_node = current_node.left_node
            else:
                if current_node.right_node is None:
                    new_node.parent = current_node
                    current_node.right_node = new_node
                    break
                else:
                    current_node = current_node.right_node

    def remove(self, target_node):
        """remove: Remove a node from the tree... This is a problem function and is quite honestly
        hot garbage! And im fairly certain there is a rather catastrophic logic error 'somewhere'???

        :param target_node: Node which you want to remove from the tree
        """
        assert isinstance(target_node, Node)
        child_count = self._count_children(target_node)
```

```
if child_count == 0:
    if self.root == target_node:
        self.root = None
    else:
        self._remove_leaf(target_node)
elif child_count == 1:
    if self.root == target_node:
        if target_node.left_node is not None:
            target_node.left_node.parent = None
            self.root = target_node.left_node
        elif target_node.right_node is not None:
            target_node.right_node.parent = None
            self.root = target_node.right_node
    else:
        if target_node.left_node is not None:
            parent = target_node.parent
            target_node.left_node.parent = parent

            if parent.left_node == target_node:
                parent.left_node = target_node.left_node

            if parent.right_node == target_node:
                parent.right_node = target_node.left_node

        elif target_node.right_node is not None:
            parent = target_node.parent
            target_node.right_node.parent = parent

            if parent.left_node == target_node:
                parent.left_node = target_node.right_node

            if parent.right_node == target_node:
                parent.right_node = target_node.right_node
elif child_count == 2:
    if self.root == target_node:
        if target_node.left_node is not None:
            swap_target = self._max_left()
        elif target_node.right_node is not None:
            swap_target = self._min_right()

        self._remove_leaf(swap_target)
        swap_target.parent = None
        swap_target.left_node = target_node.left_node
        swap_target.right_node = target_node.right_node
        self.root = swap_target

        if target_node.left_node is not None:
            target_node.left_node.parent = swap_target
        if target_node.right_node is not None:
```

```
                    target_node.right_node.parent = swap_target
            else:
                if target_node < self.root:
                    swap_target = self._max_left()
                else:
                    swap_target = self._min_right()

                self._remove_leaf(swap_target)
                swap_target.parent = target_node.parent
                swap_target.left_node = target_node.left_node
                swap_target.right_node = target_node.right_node

                if target_node.left_node is not None:
                    target_node.left_node.parent = swap_target

                if target_node.right_node is not None:
                    target_node.right_node.parent = swap_target

                parent = target_node.parent
                if parent.left_node == target_node:
                    parent.left_node = swap_target

                if parent.right_node == target_node:
                    parent.right_node = swap_target

        target_node.left_node = None
        target_node.right_node = None
        target_node.parent = None

    def order(self, string=False, in_order=True, pre_order=False, post_order=False):
        """order: Get the binary tree in one of three orders

        :param string: Whether you want each node returned in string fromat or as a node object
        :param in_order: Return the tree values in order
        :param pre_order: Return the tree values in pre order
        :param post_order: Return the tree values in post order
        """
        if self.root is None:
            return list()

        if in_order:
            return self._in_order(string)
        elif pre_order:
            return self._pre_order(string)
        elif post_order:
            return self._post_order(string)

    def find(self, target):
        """find: Uses '_find' to traverse the tree to find 'target'
        :param target: Node which you want to find
        """
        def _find(target, current_node, contents):
            if current_node.left_node is not None:
                _find(target, current_node.left_node, contents)
            if current_node.value == target:
                contents.append(current_node)
            if current_node.right_node is not None:
                _find(target, current_node.right_node, contents)
            return contents

        return _find(target, self.root, [])

    def _in_order(self, string):
        def __in_order(string, current_node, contents):
            if current_node.left_node is not None:
                __in_order(string, current_node.left_node, contents)
            if string:
                contents.append(str(current_node))
            else:
                contents.append(current_node)
            if current_node.right_node is not None:
                __in_order(string, current_node.right_node, contents)
            return contents

        return __in_order(string, self.root, [])

    def _pre_order(self, string):
        def __pre_order(string, current_node, contents):
            if string:
                contents.append(str(current_node))
            else:
                contents.append(current_node)
            if current_node.left_node is not None:
                __pre_order(string, current_node.left_node, contents)
            if current_node.right_node is not None:
                __pre_order(string, current_node.right_node, contents)
            return contents

        return __pre_order(string, self.root, [])

    def _post_order(self, string):
        def __post_order(string, current_node, contents):
            if current_node.left_node is not None:
                __post_order(string, current_node.left_node, contents)
            if current_node.right_node is not None:
                __post_order(string, current_node.right_node, contents)
            if string:
                contents.append(str(current_node))
            else:
                contents.append(current_node)
            return contents

        return __post_order(string, self.root, [])

    def _max_left(self):
        """_max_left: Gets the node with the maximum value from the
        left side of the tree

        :return Node: Max from the left
        """
        current_node = self.root.left_node
        while current_node:
            if current_node.right_node is not None:
                current_node = current_node.right_node
            else:
                break
        return current_node

    def _min_right(self):
        """_min_right: Finds the minimum value from the right side of the tree

        :return Node: Min from right side of the tree
        """
        current_node = self.root.right_node
        while current_node:
            if current_node.left_node is not None:
                current_node = current_node.left_node
            else:
                break
        return current_node

    @classmethod
    def _count_children(cls, target_node):
        """_count_children: Counts how many children there is to any node

        :param target_node: Node which you want to find out how many children it has
        :return int: Value between 0 and 2 depending on how many children
        """
        assert isinstance(target_node, Node)
        count = 0
        if target_node.left_node is not None:
            count += 1
        if target_node.right_node is not None:
            count += 1
        return count
    @classmethod
    def _remove_leaf(cls, target_node):
        """_remove_leaf: Remove a node which has no children

        :param target_node: Node to be removed from the tree
        """
        assert isinstance(target_node, Node)
        parent = target_node.parent
        if parent.left_node == target_node:
            parent.left_node = None
        elif parent.right_node == target_node:
            parent.right_node = None

    def __str__(self):
        return str(self.order(True))
```

database.py

```
#!/usr/bin/python3
""" database.py: Simple database class
"""



from binary_tree import BinaryTree
from node import Node



class Database:
    """Database: Very limited database class using binary trees to store
    date however irl I belive you would use B+ Trees and because they fan out more
    which is what is used in database software such as 'SQLite3' and filesystems
    """
    def __init__(self, students):
        """__init__

        :param students: List of studets to be added to the database
        """
        self.data = {}

        for student in students:
            for value in student.data:
                new_node = Node((student.data[value], student))
                if value in self.data:
                    self.data[value].insert(new_node)
                else:
                    self.data[value] = BinaryTree()
                    self.data[value].insert(new_node)

    def find(self, target, where):
        """find: Search the database. Has simmilar structure as 'SQLite3'

        :param target: What you want to find e.g '1', 'John'
        :param where: Where you want to search e.g 'unique_id', 'name'
        """
        if where not in self.data:
            return
        if target == '*' or target == 'all':
            return self.data[where].order()

        assert self.data[where].type == type(target)
        return self._convert_node_list(self.data[where].find(target))

    def remove_by_id(self, target_id):
        """remove_by_id: Remove something from the database by id

        :param target_id: Id of the student you want to remove
        """
        for key in self.data:
            for node in self.data[key].order():
                if node.owner.data['unique_id'] == target_id:
                    self.data[key].remove(node)

    def update(self, target, where, content):
        """update: Update the records of a student in the database

        :param target: What you want to find from the database
        :param where: Where you want to search
        :param content: What you want to replace is with
        """
        if where == 'unique_id':
            raise ValueError('Cannot update the unique_id')
        assert self.data[where].type == type(content)
        update_list = self.find(target, where)
        for student in update_list:
            student.data[where] = content

    def list(self, where):
        """list: Like 'select *' in 'SQLite3' in the way that is display
        all records from 'where'

        :param where: Where you want to show the records for
        """
        return self.data[where].order(True)

    @classmethod
    def _convert_node_list(cls, node_list):
        """_convert_node_list: Convert a list of nodes into a list of their owners
        :param node_list: List to change
        """
        return [node.owner for node in node_list]
```

<u>database_unit_test.py</u>

```
#!/usr/bin/python3

import unittest
import datetime
from database import Database
from student import Student
from address import Address

class UnitTest(unittest.TestCase):
    def setUp(self):
        self.student1 = Student(1, "Ryan", datetime.date(1978, 1, 12), Address(104, 'Main Street'), datetime.date(2017, 3, 9), "220CT", True)
        self.student2 = Student(2, "Devin", datetime.date(2000, 1, 12), Address(10, 'Station Road'), datetime.date(2017, 3, 9), "121COM", False)
        self.student3 = Student(3, "Rob", datetime.date(2002, 4, 2), Address(1, 'Lunch Lane'), datetime.date(2017, 3, 9), "290COM", True)
        self.student4 = Student(4, "Ellen", datetime.date(1997, 1, 12), Address(23, 'Lovelace Avenue'), datetime.date(2017, 3, 9), "290COM", False)
        self.student5 = Student(5, "Taylor", datetime.date(1995, 5, 9), Address(3, 'Judas Lane'), datetime.date(2017, 3, 9), "220CT", True)
        self.students = [self.student3, self.student2, self.student4, self.student1, self.student5]
        self.db = Database(self.students)

    def test_finding_student_by_id(self):
        correct = [self.student3]
        self.assertEqual(self.db.find(3, 'unique_id'), correct)

    def test_find_and_update_by_id(self):
        self.db.update('220CT', 'class_id', '210CT')
        self.assertEqual(self.student1.data['class_id'], '210CT')
        self.assertEqual(self.student5.data['class_id'], '210CT')

    def test_finding_student_by_class(self):
        correct = [self.student1, self.student5]
        self.assertEqual(self.db.find('220CT', 'class_id'), correct)

    def test_list_names_in_lex_order(self):
        correct = ['Devin', 'Ellen', 'Rob', 'Ryan', 'Taylor']
        self.assertEqual(self.db.list('name'), correct)

    def test_list_all_postgraduates(self):
        graduatedStudents = self.db.find(True, 'postgraduate')
        correct = [self.student1, self.student3, self.student5]
        for answer in correct:
            self.assertTrue(answer in graduatedStudents)

    def test_list_undergrads_by_class_in_lex_order(self):
        lex = []
        students = self.db.find('all', 'name')
        classes = self.db.find('all', 'class_id')
        for i in range(len(students)):
            for j in range(len(classes)):
                if students[i].owner == classes[j].owner:
                    lex.append('{0}: {1}'.format(classes[j], students[i]))
        self.assertEqual(lex, ['121COM: Devin', '290COM: Ellen', '290COM: Rob', '220CT: Ryan',
'220CT: Taylor'])

    def test_remove_by_id(self):
        student1 = Student(1, "Ryan", datetime.date(1978, 1, 12), Address(104, 'Main Street'),
datetime.date(2017, 2, 9), '220CT', True)
        student2 = Student(2, "Devin", datetime.date(2000, 1, 12), Address(10, 'Station Road'),
datetime.date(2013, 3, 9), '210CT', False)
        student3 = Student(3, "Rob", datetime.date(2002, 4, 2), Address(1, 'Lunch Lane'),
datetime.date(2017, 3, 4), '210CT', True)
        student4 = Student(4, "Ellen", datetime.date(1997, 1, 12), Address(1, 'Lunch Lane'),
datetime.date(2017, 3, 9), '290COM', False)
        student5 = Student(5, "Taylor", datetime.date(1995, 5, 9), Address(3, 'Judas Lane'),
datetime.date(2017, 4, 9), '220CT', True)
        students = [student5, student3, student4, student2, student1]
        db = Database(students)
        db.remove_by_id(1)
        self.assertEqual(db.data['unique_id'].order(True), ['2', '3', '4', '5'])
        self.assertEqual(db.data['name'].order(True), ['Devin', 'Ellen', 'Rob', 'Taylor'])
        self.assertEqual(db.data['date_of_birth'].order(True), ['1995-05-09', '1997-01-12', '2000-01-12',
'2002-04-02'])
        self.assertEqual(db.data['address'].order(True), ['1 Lunch Lane', '1 Lunch Lane', '3 Judas Lane', '10
Station Road'])
        self.assertEqual(db.data['enrolment_date'].order(True), ['2013-03-09', '2017-03-04', '2017-03-09',
'2017-04-09'])
        self.assertEqual(db.data['class_id'].order(True), ['210CT', '210CT', '220CT', '290COM'])
        self.assertEqual(db.data['postgraduate'].order(True), ['False', 'False', 'True', 'True'])

    def test_remove_by_postgraduate(self):
        student1 = Student(1, "Ryan", datetime.date(1978, 1, 12), Address(104, 'Main Street'),
datetime.date(2017, 2, 9), '220CT', True)
        student2 = Student(2, "Devin", datetime.date(2000, 1, 12), Address(10, 'Station Road'),
datetime.date(2013, 3, 9), '210CT', False)
        student3 = Student(3, "Rob", datetime.date(2002, 4, 2), Address(1, 'Lunch Lane'),
datetime.date(2017, 3, 4), '210CT', True)
        student4 = Student(4, "Ellen", datetime.date(1997, 1, 12), Address(1, 'Lunch Lane'),
datetime.date(2017, 3, 9), '290COM', False)
```

```
        student5 = Student(5, "Taylor", datetime.date(1995, 5, 9), Address(3, 'Judas Lane'),
datetime.date(2017, 4, 9), '220CT', True)
        students = [student5, student4, student3, student2, student1]
        db = Database(students)

        post_grads = db.find(True, 'postgraduate')
        db.remove_by_id(post_grads[0].data['unique_id'])

        self.assertEqual(db.data['unique_id'].order(True), ['1', '2', '3', '4'])
        self.assertEqual(db.data['name'].order(True), ['Devin', 'Ellen', 'Rob', 'Ryan'])
        self.assertEqual(db.data['date_of_birth'].order(True), ['1978-01-12', '1997-01-12', '2000-01-12',
'2002-04-02'])
        self.assertEqual(db.data['address'].order(True), ['1 Lunch Lane', '1 Lunch Lane', '10 Station Road',
'104 Main Street'])
        self.assertEqual(db.data['enrolment_date'].order(True), ['2013-03-09', '2017-02-09', '2017-03-04',
'2017-03-09'])
        self.assertEqual(db.data['class_id'].order(True), ['210CT', '210CT', '220CT', '290COM'])
        self.assertEqual(db.data['postgraduate'].order(True), ['False', 'False', 'True', 'True'])

        db.remove_by_id(post_grads[1].data['unique_id'])

        self.assertEqual(db.data['unique_id'].order(True), ['1', '2', '4'])
        self.assertEqual(db.data['name'].order(True), ['Devin', 'Ellen', 'Ryan'])
        self.assertEqual(db.data['date_of_birth'].order(True), ['1978-01-12', '1997-01-12', '2000-01-12'])
        self.assertEqual(db.data['address'].order(True), ['1 Lunch Lane', '10 Station Road', '104 Main
Street'])
        self.assertEqual(db.data['enrolment_date'].order(True), ['2013-03-09', '2017-02-09', '2017-03-09'])
        self.assertEqual(db.data['class_id'].order(True), ['210CT', '220CT', '290COM'])
        self.assertEqual(db.data['postgraduate'].order(True), ['False', 'False', 'True'])

        db.remove_by_id(post_grads[2].data['unique_id'])
        self.assertEqual(db.data['unique_id'].order(True), ['2', '4'])
        self.assertEqual(db.data['name'].order(True), ['Devin', 'Ellen'])
        self.assertEqual(db.data['date_of_birth'].order(True), ['1997-01-12', '2000-01-12'])
        self.assertEqual(db.data['address'].order(True), ['1 Lunch Lane', '10 Station Road'])
        self.assertEqual(db.data['enrolment_date'].order(True), ['2013-03-09', '2017-03-09'])
        self.assertEqual(db.data['class_id'].order(True), ['210CT', '290COM'])
        self.assertEqual(db.data['postgraduate'].order(True), ['False', 'False'])


if __name__ == '__main__':
    unittest.main()
```

main.py

```
#!/usr/bin/python3

import datetime
from database import Database
from student import Student
from address import Address

def main():
    student1 = Student(1, "Ryan", datetime.date(1978, 1, 12), Address(104, 'Main Street'),
datetime.date(2017, 2, 9), '220CT', True)
    student2 = Student(2, "Devin", datetime.date(2000, 1, 12), Address(10, 'Station Road'),
datetime.date(2013, 3, 9), '210CT', False)
    student3 = Student(3, "Rob", datetime.date(2002, 4, 2), Address(1, 'Lunch Lane'),
datetime.date(2017, 3, 4), '210CT', True)
    student4 = Student(4, "Ellen", datetime.date(1997, 1, 12), Address(1, 'Lunch Lane'),
datetime.date(2017, 3, 9), '290COM', False)
    student5 = Student(5, "Taylor", datetime.date(1995, 5, 9), Address(3, 'Judas Lane'),
datetime.date(2017, 4, 9), '220CT', True)
    students = [student5, student4, student3, student2, student1]
    db = Database(students)

    #  Found student by id in this case it will be a list containing the reference to 'student3'
    print(db.find(3, 'unique_id'))


main()
```

node.py

```
#!/usr/bin/python3
""" node.py: Contains the node class which is used in the simple database and binary trees
"""



class Node:
    """Node: Simple node class used in the binary trees
    """
    def __init__(self, value):
        self.value = value[0]
        self.owner = value[1]
        self.parent = None
        self.left_node = None
        self.right_node = None

    def __ne__(self, other):
        try:
            return self.value != other.value
        except AttributeError:
            return True

    def __lt__(self, other):
        return self.value < other.value
    def __gt__(self, other):
        return self.value > other.value

    def __str__(self):
        return str(self.value)
```

student.py

```
#!/usr/bin/python3


import datetime
from address import Address


class Student:
    """Student: Class representing a student
    """
    def __init__(self, unique_id, name, date_of_birth, address, enrolment_date, class_id, postgraduate):
        """__init__

        :param unique_id: ID of a student
        :param name: Name of the student
        :param date_of_birth: The sutdents DOB
        :param address: Where they live
        :param enrolment_date: When they enrolled
        :param class_id: What class they are in
        :param postgraduate: Whether they have graduated
        """
        assert isinstance(unique_id, int)
        assert isinstance(name, str)
        assert isinstance(date_of_birth, datetime.date)
        assert isinstance(address, Address)
        assert isinstance(enrolment_date, datetime.date)
        assert isinstance(class_id, str)
        assert isinstance(postgraduate, bool)

        self.data = {}
        self.data['unique_id'] = unique_id
        self.data['name'] = name
        self.data['date_of_birth'] = date_of_birth
        self.data['address'] = address
        self.data['enrolment_date'] = enrolment_date
        self.data['class_id'] = class_id
        self.data['postgraduate'] = postgraduate

    def __str__(self):
        return str(self.data['name'])
```

tree_unit_test.py

```
#!/usr/bin/python3

import unittest
from binary_tree import BinaryTree
from node import Node

class UnitTest(unittest.TestCase):
    def test_insert_root(self):
        tree = BinaryTree()
        node1 = Node((1, None))
        tree.insert(node1)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, None)
        self.assertEqual(node1.right_node, None)

    def test_insert_root_with_one_child(self):
        tree = BinaryTree()
        node1 = Node((1, None))
        node2 = Node((2, None))
        tree.insert(node1)
        tree.insert(node2)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, None)
        self.assertEqual(node1.right_node, node2)
        self.assertEqual(node2.parent, node1)
        self.assertEqual(node2.left_node, None)
        self.assertEqual(node2.right_node, None)

    def test_insert_three(self):
        tree = BinaryTree()
        node1 = Node((3, None))
        node2 = Node((2, None))
        node3 = Node((4, None))
        tree.insert(node1)
        tree.insert(node2)
        tree.insert(node3)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, node2)
        self.assertEqual(node1.right_node, node3)
        self.assertEqual(node2.parent, node1)
        self.assertEqual(node2.left_node, None)
        self.assertEqual(node2.right_node, None)
        self.assertEqual(node3.parent, node1)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)

    def test_min_right(self):
        tree = BinaryTree()
        node1 = Node((1, None))
        node2 = Node((5, None))
        node3 = Node((7, None))
        node4 = Node((4, None))
        tree.insert(node1)
        tree.insert(node2)
        tree.insert(node3)
        tree.insert(node4)
        self.assertEqual(tree._min_right(), node4)

    def test_count_children(self):
        tree = BinaryTree()
        node1 = Node((1, None))
        node2 = Node((5, None))
        node3 = Node((7, None))
        node4 = Node((4, None))
        tree.insert(node1)
        tree.insert(node2)
        tree.insert(node3)
        tree.insert(node4)
        self.assertEqual(tree._count_children(node2), 2)
        self.assertEqual(tree._count_children(node1), 1)

    def test_remove_leaf(self):
        tree = BinaryTree()
        node1 = Node((2, None))
        node2 = Node((1, None))
        tree.insert(node1)
        tree.insert(node2)
        self.assertEqual(tree.order(), [node2, node1])
        tree.remove(node2)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, None)
        self.assertEqual(node1.right_node, None)
        self.assertEqual(node2.parent, None)
        self.assertEqual(node2.left_node, None)
        self.assertEqual(node2.right_node, None)
        self.assertEqual(tree.order(), [node1])

    def test_remove_1_right_child(self):
        tree = BinaryTree()
        node1 = Node((5, None))
        node2 = Node((3, None))
        node3 = Node((4, None))
        tree.insert(node1)
        tree.insert(node2)
        tree.insert(node3)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.left_node, node2)
        self.assertEqual(node1.right_node, None)
        self.assertEqual(node2.parent, node1)
        self.assertEqual(node2.left_node, None)
        self.assertEqual(node2.right_node, node3)
        self.assertEqual(node3.parent, node2)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(tree.order(), [node2, node3, node1])
        tree.remove(node2)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.left_node, node3)
        self.assertEqual(node1.right_node, None)
        self.assertEqual(node3.parent, node1)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(tree.order(), [node3, node1])

    def test_remove_1_left_child(self):
        tree = BinaryTree()
        node1 = Node((5, None))
        node2 = Node((3, None))
        node3 = Node((2, None))
        tree.insert(node1)
        tree.insert(node2)
        tree.insert(node3)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, node2)
        self.assertEqual(node1.right_node, None)
        self.assertEqual(node2.parent, node1)
        self.assertEqual(node2.left_node, node3)
        self.assertEqual(node2.right_node, None)
        self.assertEqual(node3.parent, node2)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(tree.order(), [node3, node2, node1])
        tree.remove(node2)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, node3)
        self.assertEqual(node1.right_node, None)
        self.assertEqual(node2.parent, None)
        self.assertEqual(node2.left_node, None)
        self.assertEqual(node2.right_node, None)
        self.assertEqual(node3.parent, node1)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(tree.order(), [node3, node1])

    def test_remove_2_children(self):
        tree = BinaryTree()
        node1 = Node((5, None))
        node2 = Node((7, None))
        node3 = Node((8, None))
        node4 = Node((6, None))
        tree.insert(node1)
        tree.insert(node2)
        tree.insert(node3)
        tree.insert(node4)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.left_node, None)
        self.assertEqual(node1.right_node, node2)
        self.assertEqual(node2.parent, node1)
        self.assertEqual(node2.left_node, node4)
        self.assertEqual(node2.right_node, node3)
        self.assertEqual(node4.parent, node2)
        self.assertEqual(node3.parent, node2)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(node4.left_node, None)
        self.assertEqual(node4.right_node, None)
        self.assertEqual(tree.order(), [node1, node4, node2, node3])
        tree.remove(node2)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.left_node, None)
        self.assertEqual(node1.right_node, node4)
        self.assertEqual(node2.parent, None)
        self.assertEqual(node2.left_node, None)
        self.assertEqual(node2.right_node, None)
        self.assertEqual(node4.parent, node1)
        self.assertEqual(node3.parent, node4)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(node4.left_node, None)
        self.assertEqual(node4.right_node, node3)
        self.assertEqual(tree.order(), [node1, node4, node3])

    def test_remove_2_children(self):
        tree = BinaryTree()
        node1 = Node((5, None))
        node2 = Node((3, None))
        node3 = Node((2, None))
        node4 = Node((4, None))
        tree.insert(node1)
        tree.insert(node2)
        tree.insert(node3)
        tree.insert(node4)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.left_node, node2)
        self.assertEqual(node1.right_node, None)
        self.assertEqual(node2.parent, node1)
        self.assertEqual(node2.left_node, node3)
        self.assertEqual(node2.right_node, node4)
        self.assertEqual(node4.parent, node2)
        self.assertEqual(node3.parent, node2)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(node4.left_node, None)
        self.assertEqual(node4.right_node, None)
        self.assertEqual(tree.order(), [node3, node2, node4, node1])
        tree.remove(node2)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.left_node, node4)
        self.assertEqual(node1.right_node, None)
        self.assertEqual(node2.parent, None)
        self.assertEqual(node2.left_node, None)
        self.assertEqual(node2.right_node, None)
        self.assertEqual(node4.parent, node1)
        self.assertEqual(node3.parent, node4)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(node4.left_node, node3)
        self.assertEqual(node4.right_node, None)
        self.assertEqual(tree.order(), [node3, node4, node1])

    def test_remove_2_children(self):
        tree = BinaryTree()
        node1 = Node((10, None))
        node2 = Node((20, None))
        node3 = Node((15, None))
        node4 = Node((25, None))
        node5 = Node((5, None))
        node6 = Node((7, None))
        node7 = Node((3, None))
        node8 = Node((2, None))
        tree.insert(node1)
        tree.insert(node2)
        tree.insert(node3)
        tree.insert(node4)
        tree.insert(node5)
        tree.insert(node6)
        tree.insert(node7)
        tree.insert(node8)
```

```
self.assertEqual(tree.root, node1)
self.assertEqual(node1.left_node, node5)
self.assertEqual(node1.right_node, node2)
self.assertEqual(node1.parent, None)
self.assertEqual(node2.parent, node1)
self.assertEqual(node2.left_node, node3)
self.assertEqual(node2.right_node, node4)
self.assertEqual(node3.parent, node2)
self.assertEqual(node3.left_node, None)
self.assertEqual(node3.right_node, None)
self.assertEqual(node4.parent, node2)
self.assertEqual(node4.left_node, None)
self.assertEqual(node4.right_node, None)
self.assertEqual(node5.parent, node1)
self.assertEqual(node5.left_node, node7)
self.assertEqual(node5.right_node, node6)
self.assertEqual(node6.parent, node5)
self.assertEqual(node6.left_node, None)
self.assertEqual(node6.right_node, None)
self.assertEqual(node7.parent, node5)
self.assertEqual(node7.left_node, node8)
self.assertEqual(node7.right_node, None)
self.assertEqual(node8.parent, node7)
self.assertEqual(node8.left_node, None)
self.assertEqual(node8.right_node, None)
self.assertEqual(tree.order(), [node8, node7, node5, node6, node1, node3, node2, node4])
tree.remove(node5)
self.assertEqual(tree.root, node1)
self.assertEqual(node1.left_node, node6)
self.assertEqual(node1.right_node, node2)
self.assertEqual(node1.parent, None)
self.assertEqual(node2.parent, node1)
self.assertEqual(node2.left_node, node3)
self.assertEqual(node2.right_node, node4)
self.assertEqual(node3.parent, node2)
self.assertEqual(node3.left_node, None)
self.assertEqual(node3.right_node, None)
self.assertEqual(node4.parent, node2)
self.assertEqual(node4.left_node, None)
self.assertEqual(node4.right_node, None)
self.assertEqual(node5.parent, None)
self.assertEqual(node5.left_node, None)
self.assertEqual(node5.right_node, None)
self.assertEqual(node6.parent, node1)
self.assertEqual(node6.left_node, node7)
self.assertEqual(node6.right_node, None)
self.assertEqual(node7.parent, node6)
self.assertEqual(node7.left_node, node8)
self.assertEqual(node7.right_node, None)
```

```
        self.assertEqual(node8.parent, node7)
        self.assertEqual(node8.left_node, None)
        self.assertEqual(node8.right_node, None)
        self.assertEqual(tree.order(), [node8, node7, node6, node1, node3, node2, node4])

    def test_remove_root_left_child(self):
        tree = BinaryTree()
        node1 = Node((5, None))
        node2 = Node((3, None))
        node3 = Node((2, None))
        node4 = Node((4, None))
        tree.insert(node1)
        tree.insert(node2)
        tree.insert(node3)
        tree.insert(node4)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, node2)
        self.assertEqual(node1.right_node, None)
        self.assertEqual(node2.parent, node1)
        self.assertEqual(node2.left_node, node3)
        self.assertEqual(node2.right_node, node4)
        self.assertEqual(node4.parent, node2)
        self.assertEqual(node3.parent, node2)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(node4.left_node, None)
        self.assertEqual(node4.right_node, None)
        self.assertEqual(tree.order(), [node3, node2, node4, node1])
        tree.remove(node1)
        self.assertEqual(tree.root, node2)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, None)
        self.assertEqual(node1.right_node, None)
        self.assertEqual(node2.parent, None)
        self.assertEqual(node2.left_node, node3)
        self.assertEqual(node2.right_node, node4)
        self.assertEqual(node4.parent, node2)
        self.assertEqual(node3.parent, node2)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(node4.left_node, None)
        self.assertEqual(node4.right_node, None)
        self.assertEqual(tree.order(), [node3, node2, node4])

    def test_remove_root_right_child(self):
        tree = BinaryTree()
        node1 = Node((5, None))
        node2 = Node((10, None))
        node3 = Node((8, None))
        node4 = Node((12, None))
        tree.insert(node1)
        tree.insert(node2)
        tree.insert(node3)
        tree.insert(node4)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, None)
        self.assertEqual(node1.right_node, node2)
        self.assertEqual(node2.parent, node1)
        self.assertEqual(node2.left_node, node3)
        self.assertEqual(node2.right_node, node4)
        self.assertEqual(node4.parent, node2)
        self.assertEqual(node3.parent, node2)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(node4.left_node, None)
        self.assertEqual(node4.right_node, None)
        self.assertEqual(tree.order(), [node1, node3, node2, node4])
        tree.remove(node1)
        self.assertEqual(tree.root, node2)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, None)
        self.assertEqual(node1.right_node, None)
        self.assertEqual(node2.parent, None)
        self.assertEqual(node2.left_node, node3)
        self.assertEqual(node2.right_node, node4)
        self.assertEqual(node4.parent, node2)
        self.assertEqual(node3.parent, node2)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(node4.left_node, None)
        self.assertEqual(node4.right_node, None)
        self.assertEqual(tree.order(), [node3, node2, node4])

    def test_remove_root_2_children(self):
        tree = BinaryTree()
        node1 = Node((5, None))
        node2 = Node((10, None))
        node3 = Node((4, None))
        node4 = Node((12, None))
        tree.insert(node1)
        tree.insert(node2)
        tree.insert(node3)
        tree.insert(node4)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, node3)
        self.assertEqual(node1.right_node, node2)
        self.assertEqual(node2.parent, node1)
        self.assertEqual(node2.left_node, None)
        self.assertEqual(node2.right_node, node4)
        self.assertEqual(node4.parent, node2)
        self.assertEqual(node3.parent, node1)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(node4.left_node, None)
        self.assertEqual(node4.right_node, None)
        self.assertEqual(tree.order(), [node3, node1, node2, node4])
        tree.remove(node1)
        self.assertEqual(tree.root, node3)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, None)
        self.assertEqual(node1.right_node, None)
        self.assertEqual(node2.parent, node3)
        self.assertEqual(node2.left_node, None)
        self.assertEqual(node2.right_node, node4)
        self.assertEqual(node4.parent, node2)
        self.assertEqual(node3.parent, None)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, node2)
        self.assertEqual(node4.left_node, None)
        self.assertEqual(node4.right_node, None)
        self.assertEqual(tree.order(), [node3, node2, node4])

    def test_equal_values(self):
        tree = BinaryTree()
        node1 = Node((1, None))
        node2 = Node((1, None))
        node3 = Node((1, None))
        node4 = Node((1, None))
        tree.insert(node1)
        tree.insert(node2)
        tree.insert(node3)
        tree.insert(node4)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, None)
        self.assertEqual(node1.right_node, node2)
        self.assertEqual(node2.parent, node1)
        self.assertEqual(node2.left_node, None)
        self.assertEqual(node2.right_node, node3)
        self.assertEqual(node3.parent, node2)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, node4)
        self.assertEqual(node4.parent, node3)
        self.assertEqual(node4.left_node, None)
        self.assertEqual(node4.right_node, None)
        self.assertEqual(tree.order(), [node1, node2, node3, node4])
        tree.remove(node4)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, None)
        self.assertEqual(node1.right_node, node2)
        self.assertEqual(node2.parent, node1)
        self.assertEqual(node2.left_node, None)
        self.assertEqual(node2.right_node, node3)
        self.assertEqual(node3.parent, node2)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(node4.parent, None)
        self.assertEqual(node4.left_node, None)
        self.assertEqual(node4.right_node, None)
        self.assertEqual(tree.order(), [node1, node2, node3])

    def test_multiple_removes(self):
        tree = BinaryTree()
        node1 = Node((5, None))
        node2 = Node((10, None))
        node3 = Node((4, None))
        node4 = Node((12, None))
        tree.insert(node1)
        tree.insert(node2)
        tree.insert(node3)
        tree.insert(node4)
        self.assertEqual(tree.root, node1)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, node3)
        self.assertEqual(node1.right_node, node2)
        self.assertEqual(node2.parent, node1)
        self.assertEqual(node2.left_node, None)
        self.assertEqual(node2.right_node, node4)
        self.assertEqual(node4.parent, node2)
        self.assertEqual(node3.parent, node1)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, None)
        self.assertEqual(node4.left_node, None)
        self.assertEqual(node4.right_node, None)
        self.assertEqual(tree.order(), [node3, node1, node2, node4])
        tree.remove(node1)
        self.assertEqual(tree.root, node3)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, None)
        self.assertEqual(node1.right_node, None)
        self.assertEqual(node2.parent, node3)
        self.assertEqual(node2.left_node, None)
        self.assertEqual(node2.right_node, node4)
        self.assertEqual(node4.parent, node2)
        self.assertEqual(node3.parent, None)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, node2)
        self.assertEqual(node4.left_node, None)
        self.assertEqual(node4.right_node, None)
        self.assertEqual(tree.order(), [node3, node2, node4])
        tree.remove(node2)
        self.assertEqual(tree.root, node3)
        self.assertEqual(node1.parent, None)
        self.assertEqual(node1.left_node, None)
        self.assertEqual(node1.right_node, None)
        self.assertEqual(node2.parent, None)
        self.assertEqual(node2.left_node, None)
        self.assertEqual(node2.right_node, None)
        self.assertEqual(node4.parent, node3)
        self.assertEqual(node3.parent, None)
        self.assertEqual(node3.left_node, None)
        self.assertEqual(node3.right_node, node4)
        self.assertEqual(node4.left_node, None)
        self.assertEqual(node4.right_node, None)
        self.assertEqual(tree.order(), [node3, node4])


if __name__ == '__main__':
    unittest.main()
```

Task 7 Part 1

graph.py

```
#!/usr/bin/python3
"""graph.py: Base graph class
"""

import copy
from node import Node


class Graph:
    """graph: Undirected unweighted graph class
    """
    def __init__(self, nodes):
        """__init__

        :param nodes: List of nodes to be added to the graph
        """
        assert isinstance(nodes, list)
        self.vertices = set()
        self.edges = {}

        for new_node in nodes:
            self.add_node(new_node)

    def is_connected(self):
        """is_connected: Function uses '_is_connected' to determine if graph is connected
        """
        visited = set()
        start = next(iter(self.vertices))

        def _is_connected(visited, start):
            """_is_connected

            :param visited: Set of visited nodes
            :param start: Node where we start if none is supplied the first one in 'vertices' will
            be used
            """
            visited.add(start)
            if len(visited) == len(self.vertices):
                return True
            for vertex in self.edges[start]:
                if vertex not in visited:
                    if _is_connected(visited, vertex):
                        return True
            return False

        return _is_connected(visited, start)

    def add_node(self, target_node):
        """add_node

        :param node: Adds a node to the graph using the helper functions '_add_edge' and
        '_add_vertex'
        """
        assert isinstance(target_node, Node)
        self._add_vertex(target_node.value)
        for connection in target_node.connections:
            self._add_edge(target_node.value, connection)

    def remove_node(self, target_node):
        """remove_node

        :param node: Removes a node from the graph using the help functions '_remove_edge'
        and '_remove_vertex'
        """
        assert isinstance(target_node, Node)
        self._remove_vertex(target_node.value)
        self._remove_edge(target_node.value)

    def find_all_paths(self):
        """find_all_paths: Finds all valid paths through the graph using DFS

        :return list: all paths is graph
        """
        all_paths = []

        def _find_path(start, end, visited, path):
            visited[start] = True
            path.append(start)
            if start == end:
                all_paths.append(copy.copy(path))
            else:
                for vertex in self.edges[start]:
                    if visited[vertex] is False:
                        _find_path(vertex, end, visited, path)
            path.pop()
            visited[start] = False

        for start in self.vertices:
            for end in self.vertices:
                _find_path(start, end, dict.fromkeys(self.vertices, False), [])
        return all_paths

    def get_hamiltonian_cycles(self):
        """get_hamiltonian_cycles: Uses output of 'find_all_paths' to construct a list of
        hamiltonian cycles

        :return list: List of hamiltonian cycles
        """
        cycles = []

        def _find_cycles():
            all_paths = self.find_all_paths()
            for path in all_paths:
                if set(path) == self.vertices:
                    start = path[0]
                    end = path[-1]
                    if start in self.edges[end]:
                        path.append(path[0])
                        cycles.append(path)

        _find_cycles()
        return cycles

    def _remove_vertex(self, value):
        self.vertices.remove(value)

    def _remove_edge(self, value):
        del self.edges[value]
        for key in self.edges:
            self.edges[key].remove(value)

    def _add_vertex(self, value):
        self.vertices.add(value)

    def _add_edge(self, vertex_a, vertex_b):
        if vertex_a not in self.vertices:
            self._add_vertex(vertex_a)
        if vertex_b not in self.vertices:
            self._add_vertex(vertex_b)
        if vertex_a not in self.edges:
            self.edges[vertex_a] = set()
        self.edges[vertex_a].add(vertex_b)

        if vertex_b not in self.edges:
            self.edges[vertex_b] = set()
        self.edges[vertex_b].add(vertex_a)
```

main.py

```
#!/usr/bin/python3
"""main.py: Code to show the is_connected function working
"""

from graph import Graph
from node import Node


def main():
    """main: Example to show testing if a graph is connected or not
    """
    node1 = Node(1, [2, 4])
    node2 = Node(2, [1, 3])
    node3 = Node(3, [2])
    node4 = Node(4, [1])
    graph1 = Graph([node1, node2, node3, node4])

    if graph1.is_connected():
        print("The graph is connected")
    else:
        print("The graph is not connected")


main()
```

node.py

```
#!/usr/bin/python3
"""node.py: Contains node class which will allow the creation of undirected nodes
"""


class Node:
    """Node: Node class representing nodes on a graph
    """
    def __init__(self, value, connections):
        """__init__

        :param value: Value of the node can only be a 'str' or 'int'
        :param connections: List of the nodes connections base of other nodes value e.g [1, 2, 4] or
        ['a', 'b', 'c']
        """
        assert isinstance(value, (int, str))
        assert isinstance(connections, list)
        self.value = value
        self.connections = connections
```

unit_test.py

```
#!/usr/bin/python3
"""unit_test.py: Testing to make sure that the unweighted undirected graph class
functions are working
"""

import unittest
from node import Node
from graph import Graph


class UnitTest(unittest.TestCase):
    """UnitTest"""

    def test_add_node(self):
        """test_add_node: Testing adding a new node to the graph
        """
        node_1 = Node(2, [1, 3])
        graph = Graph([])
        self.assertEqual(graph.vertices, set({}))
        self.assertEqual(graph.edges, {})
        graph.add_node(node_1)
        self.assertEqual(graph.vertices, set({1, 2, 3}))
        self.assertEqual(graph.edges, {1: set({2}), 2: set({1, 3}), 3: set({2})})
    def test_remove_node(self):
        """test_remove_node: Testing removing of an old node
        """
        node_1 = Node(1, [3])
        graph = Graph([node_1])
        graph.remove_node(node_1)
        self.assertEqual(graph.vertices, set({3}))
        self.assertEqual(graph.edges, {3: set({})})

    def test_is_connected_false(self):
        """test_is_connected_false: Test that is_connected works to find out if a
        graph is not connected
        """
        node_1 = Node(1, [2])
        node_2 = Node(2, [1])
        node_3 = Node(3, [4])
        node_4 = Node(4, [3])
        graph = Graph([node_1, node_2, node_3, node_4])
        self.assertFalse(graph.is_connected())

    def test_is_connected_true(self):
        """test_is_connected_true: Test that is_connected works to find out if a
        graph is connected
        """
        node_1 = Node(1, [2, 4])
        node_2 = Node(2, [4])
        node_3 = Node(3, [2, 1])
        graph = Graph([node_1, node_2, node_3])
        self.assertTrue(graph.is_connected())


if __name__ == '__main__':
    unittest.main()
```

Task 7 Part 2

main.py

```
#!/usr/bin/python3
"""main.py: Code to show 'shortest_path' and 'longest_path' working
"""

from weighted_graph import WeightedGraph
from weighted_node import WeightedNode


def main():
    """main: Example case for finding longest and shortest path
    """
    node_0 = WeightedNode(0, [1, 4], {(0, 1): 1, (0, 4): 11})
    node_1 = WeightedNode(1, [2], {(1, 2): 2})
    node_2 = WeightedNode(2, [3], {(2, 3): 3})
    node_3 = WeightedNode(3, [], {})
    node_4 = WeightedNode(4, [5], {(4, 5): 14})
    node_5 = WeightedNode(5, [3], {(5, 3): 12})
    graph = WeightedGraph([node_0, node_1, node_2, node_3, node_4, node_5])

    print('The longest path is {0}'.format(graph.longest_path(0, 3)))
    print('The shortest path is {0}'.format(graph.shortest_path(0, 3)))


main()
```

unit_test.py

```
#!/usr/bin/python3
"""unit_test.py: Testing the 'weighted_graph' and 'weighted_node' classes
"""

import unittest
from weighted_graph import WeightedGraph
from weighted_node import WeightedNode


class UnitTest(unittest.TestCase):
    """UnitTest"""
    def test_add_node(self):
        """test_add_node: Test adding a new node
        """
        node_1 = WeightedNode(1, [2], {(1, 2): 2})
        node_2 = WeightedNode(2, [1], {(2, 1): 2})
        node_3 = WeightedNode(3, [1, 2], {(3, 1): 2, (3, 2): 4})
        graph = WeightedGraph([node_1, node_2])
        graph.add_node(node_3)

        self.assertEqual(graph.vertices, set({1, 2, 3}))
        self.assertEqual(graph.edges, {1: set({2}), 2: set({1}), 3: set({1, 2})})
        self.assertEqual(graph.weights, {(1, 2): 2, (2, 1): 2, (3, 1): 2, (3, 2): 4})

    def test_remove(self):
        """test_remove: Test removing a node
        """
        node_1 = WeightedNode(1, [2], {(1, 2): 2})
        node_2 = WeightedNode(2, [1], {(2, 1): 2})
        graph = WeightedGraph([node_1, node_2])
        graph.remove_node(node_1)

        self.assertEqual(graph.vertices, set({2}))
        self.assertEqual(graph.edges, {2: set({})})
        self.assertEqual(graph.weights, {})

    def test_topological_sort(self):
        """test_topological_sort: There is a picture in this folder showing the visual
        representation of this graph. Testing generating a topologically sorted list
        of graph vertices
        """
        node_0 = WeightedNode(5, [11], {(5, 11): 1})
        node_1 = WeightedNode(11, [2, 9, 10], {(11, 2): 1, (11, 9): 1, (11, 10): 1})
        node_2 = WeightedNode(2, [], {})
        node_3 = WeightedNode(7, [8, 11], {(7, 8): 1, (7, 11): 1})
        node_4 = WeightedNode(8, [9], {(8, 9): 1})
        node_5 = WeightedNode(9, [], {})
        node_6 = WeightedNode(3, [8, 10], {(3, 8): 1, (3, 10): 1})
        node_7 = WeightedNode(10, [], {})
        graph = WeightedGraph([node_0, node_1, node_2, node_3, node_4, node_5, node_6, node_7])
        self.assertEqual(graph.topological_sort(), [7, 5, 11, 3, 10, 8, 9, 2])

    def test_topological_sort_error(self):
        """test_topological_sort_error: Make sure that 'topological_sort' raises the correct error
        """
        node_0 = WeightedNode(1, [2], {(1, 2): 1})
        node_1 = WeightedNode(2, [1], {(2, 1): 1})
        graph = WeightedGraph([node_0, node_1])
        with self.assertRaises(TypeError):
            graph.topological_sort()

    def test_shortest_path(self):
        """test_shortest_path: Test that the shortest math is correctly generated using
        the '_dijkstra' function
        """
        node_0 = WeightedNode(0, [1, 4], {(0, 1): 1, (0, 4): 11})
        node_1 = WeightedNode(1, [2], {(1, 2): 2})
        node_2 = WeightedNode(2, [3], {(2, 3): 3})
        node_3 = WeightedNode(3, [], {})
        node_4 = WeightedNode(4, [5], {(4, 5): 14})
        node_5 = WeightedNode(5, [3], {(5, 3): 12})
        graph = WeightedGraph([node_0, node_1, node_2, node_3, node_4, node_5])
        self.assertEqual(graph.shortest_path(0, 3), 'Path: [0, 1, 2, 3]\nDistance traveled: 6')

        node_0 = WeightedNode(5, [11, 9], {(5, 11): 4, (5, 9): 1})
        node_1 = WeightedNode(11, [2, 9, 10], {(11, 2): 6, (11, 9): 2, (11, 10): 9})
        node_2 = WeightedNode(2, [5], {(2, 5): 2})
        node_3 = WeightedNode(7, [8, 11], {(7, 8): 2, (7, 11): 5})
        node_4 = WeightedNode(8, [9], {(8, 9): 5})
        node_5 = WeightedNode(9, [2], {(9, 2): 4})
        node_6 = WeightedNode(3, [8, 10], {(3, 8): 2, (3, 10): 7})
        node_7 = WeightedNode(10, [9], {(10, 9): 4})
        graph = WeightedGraph([node_0, node_1, node_2, node_3, node_4, node_5, node_6, node_7])
        self.assertEqual(graph.shortest_path(5, 9), 'Path: [5, 9]\nDistance traveled: 1')

    def test_longest_path(self):
        """test_longest_path: Test that the longest path is correctly generated using
        the '_bellman_ford' function
        """
        node_0 = WeightedNode(0, [1, 4], {(0, 1): 1, (0, 4): 11})
        node_1 = WeightedNode(1, [2], {(1, 2): 2})
        node_2 = WeightedNode(2, [3], {(2, 3): 3})
        node_3 = WeightedNode(3, [], {})
        node_4 = WeightedNode(4, [5], {(4, 5): 14})
        node_5 = WeightedNode(5, [3], {(5, 3): 12})
        graph = WeightedGraph([node_0, node_1, node_2, node_3, node_4, node_5])
        self.assertEqual(graph.longest_path(0, 3), 'Path: [0, 4, 5, 3]\nDistance traveled: 37')


if __name__ == '__main__':
    unittest.main()
```

weighted_graph.py

```
#!/usr/bin/python3
"""weighted_graph.py: 'weighted_graph' class which allows for pathfinding
"""

import math
from graph import Graph
from weighted_node import WeightedNode


class WeightedGraph(Graph):
    """weighted_graph: Directed acyclic graph class which inherits from Graph class
    """

    def __init__(self, nodes):
        """__init__

        :param nodes: List of nodes to be added to the graph
        """
        self.weights = {}
        Graph.__init__(self, nodes)

    def add_node(self, target_node):
        """add_node

        :param target_node: Overridden 'add_node' function to include adding of 'weights'
        """
        assert isinstance(target_node, WeightedNode)
    self._add_vertex(target_node.value)
    self._add_weights(target_node.weights)
    for connection in target_node.connections:
        self._add_edge(target_node.value, connection)

def remove_node(self, target_node):
    """remove_node

    :param target_node: Overridden 'remove_node' function to include removal of 'weights'
    """
    assert isinstance(target_node, WeightedNode)
    self._remove_vertex(target_node.value)
    self._remove_edge(target_node.value)
    self._remove_weights(target_node.value, target_node.weights)

def topological_sort(self):
    """topological_sort: Function to topologically sort a graph
    https://en.wikipedia.org/wiki/Topological_sorting

    :returns: List representing the graph nodes in topologically sorted order
    """
    stack = []
    visited = set()
    top_order = []

    def _visit(node):
        if node in visited:
            return
        if node in stack:
            raise TypeError('Graph contains a cycle')
        stack.append(node)
        if node in self.edges:
            for neighbour in self.edges[node]:
                _visit(neighbour)
        stack.pop()
        top_order.insert(0, node)
        visited.add(node)

    while visited != self.vertices:
        for node in self.vertices:
            _visit(node)
    return top_order

def longest_path(self, start, end):
    """longest_path: Uses _bellman_ford to calculate longest path by negating weights

    :param start: Integer representing the start node
    :param end: Integer representing the end node
    """
```

```
        assert isinstance(start, int)
        assert isinstance(end, int)
        distance, path = self._bellman_ford(start, end)
        if math.isinf(distance):
            raise ValueError('There is no path from {0} to {1}'.format(start, end))
        return 'Path: {0}\nDistance traveled: {1}'.format(path, distance * -1)

    def shortest_path(self, start, end):
        """shortest_path: Use _dijkstra to calculate shortest path

        :param start: Integer representing the start node
        :param end: Integer representing the end node
        """
        assert isinstance(start, int)
        assert isinstance(end, int)
        distance, path = self._dijkstra(start, end)
        if math.isinf(distance):
            raise ValueError('There is no path from {0} to {1}'.format(start, end))
        return 'Path: {0}\nDistance traveled: {1}'.format(path, distance)

    def get_shortest_hamiltonian_cycle(self):
        """get_shortest_hamiltonian_cycle: Find the shortest hamiltonian cycle in a graph

        :return list: list containing the shortest hamiltonian path if there is multiple of
        the same length they are all returned
        """
        def _path_cost(path):
            cost = 0
            for index, vertex in enumerate(path):
                try:
                    cost += self.weights[(vertex, path[index + 1])]
                except IndexError:
                    pass
            return path, cost

        costs = []
        cycles = self.get_hamiltonian_cycles()
        for cycle in cycles:
            costs.append(_path_cost(cycle))

        minimum = math.inf
        minimum_cycle = []
        for cycle in costs:
            if cycle[1] < minimum:
                minimum = cycle[1]
                minimum_cycle.append(cycle)

            elif cycle[1] == minimum:
                minimum_cycle.append(cycle)
        return minimum_cycle

    def _bellman_ford(self, start, end):
        """_bellman_ford: Use Belmon ford to calculate the longest path. This is needed because
        Dijkstra's weights must be non-negative

        :param start: Integer representing the start node
        :param end: Integer representing the end node
        :return Tuple: Tuple (shortest distance, sorted path)
        """
        for i in range(len(self.vertices)):
            assert start in self.vertices and end in self.vertices
            distances = dict.fromkeys(self.vertices, math.inf)
            predecessors = dict.fromkeys(self.vertices, None)
            distances[start] = 0

            for vertex in self.vertices and self.edges:
                for neighbour in self.edges[vertex]:
                    if distances[neighbour] > distances[vertex] + \
                            (self.weights[(vertex, neighbour)] * -1):
                        distances[neighbour] = distances[vertex] + \
                            (self.weights[(vertex, neighbour)] * -1)
                        predecessors[neighbour] = vertex

            for vertex in self.vertices and self.edges:
                for neighbour in self.edges:
                    if (vertex, neighbour) in self.weights:
                        if distances[neighbour] > distances[vertex] + \
                                (self.weights[(vertex, neighbour)] * -1):
                            raise TypeError('This graph contains a negative cycle')
            return distances[end], self._short_path(predecessors, end)

    def _dijkstra(self, start, end):
        """_dijkstra: Use dijkstra algorithm to get the shortest path

        :param start: Integer representing the start node
        :param end: Integer representing the end node
        :return Tuple: Tuple (shortest distance, sorted path)
        """
        assert start in self.vertices and end in self.vertices
        visited = set()
        distances = dict.fromkeys(list(self.vertices), math.inf)
        path = dict.fromkeys(list(self.vertices), None)
        distances[start] = 0
        while visited != self.vertices:
            vertex = min(set(distances.keys()) - visited)
            if vertex in self.edges:
                for neighbour in self.edges[vertex]:
                    test_path = distances[vertex] + self.weights[(vertex, neighbour)]
                    if test_path < distances[neighbour]:
                        distances[neighbour] = test_path
                        path[neighbour] = vertex
            visited.add(vertex)
        return distances[end], self._short_path(path, end)

    def _add_edge(self, vertex_a, vertex_b):
        if vertex_a not in self.edges:
            self.edges[vertex_a] = set()
        self.edges[vertex_a].add(vertex_b)

    def _add_weights(self, weights):
        for weight in weights:
            self.weights[weight] = weights[weight]

    def _remove_weights(self, value, weights):
        for weight in list(weights):
            del self.weights[weight]
        for weight in list(self.weights):
            if value in weight:
                del self.weights[weight]

    @classmethod
    def _short_path(cls, path, end):
        """_short_path

        :param path: Dictionary created above
        :param end: Integer representing the end node
        :return List: Shortest path
        """
        short_path = []
        node = end
        while path[node] is not None:
            short_path.insert(0, node)
            node = path[node]
        short_path.insert(0, node)
        return short_path
```

weighted_node.py

```
#!/usr/bin/python3
"""weighted_node.py: File containing WeightedNode class
"""

from node import Node


class WeightedNode(Node):
    """WeightedNode class allowing for nodes to have a weight(distance to other nodes)
    """
    def __init__(self, value, connections, weights):
        """__init__

        :param value: Value of the node can only be a 'str' or 'int'
        :param connections: List of the nodes connections base of other nodes value e.g [1, 2, 4]
         or ['a', 'b', 'c']
        :param weights: Dictionary - Keys are tuples from this node to another connection
        """
        assert isinstance(weights, dict)
        assert len(weights) == len(connections)
        for connection in connections:
            assert (value, connection) in weights

        self.weights = weights
        Node.__init__(self, value, connections)
```

Task 8 Part 1

main.py

```
#!/usr/bin/python3
"""
"""



from string_converter import levenshtein



print('Distance between "abc" and "abcd": {0}'.format(levenshtein('abc', 'abcd')))
print('Distance between "abd" and "ab": {0}'.format(levenshtein('abd', 'ab')))
print('Distance between "abd" and "abc": {0}'.format(levenshtein('abd', 'abc')))
print('Distance between "kitten" and "sitting": {0}'.format(levenshtein('kitten', 'sitting')))

print('Distance between "abc" and "abcd": {0}'.format(levenshtein('abc', 'abcd', 3, 4, 5)))
print('Distance between "abd" and "ab": {0}'.format(levenshtein('abd', 'ab', 3, 4, 5)))
print('Distance between "abd" and "abc": {0}'.format(levenshtein('abd', 'abc', 3, 5, 6)))
print('Distance between "kitten" and "sitting": {0}'.format(levenshtein('kitten',
                                                               'sitting', 3, 4, 5)))
```

string_converter.py

```
#!/usr/bin/python3
""" string_converter.py: Is the file which contains the functions to convert
one string to another as cheaply as possible following a simple set of rules
"""
```

```
def levenshtein(source, target, del_cost=1, ins_cost=1, sub_cost=1):
    """levenshtein: Function to find out the distance between two strings

    :param source: The string which you want to change
    :param target: The target for which you want to make string into
    :param del_cost:
    :param ins_cost:
    :param sub_cost:
    """
    if not target:
        return len(source)
    previous_row = range(len(target) + 1)
    for source_index, source_character in enumerate(source):
        current_row = [source_index + 1]
        for target_index, target_character in enumerate(target):
            if source_character != target_character:
                substitution_cost = sub_cost
            else:
                substitution_cost = 0
            possible_deletions = previous_row[target_index + 1] + del_cost
            possible_insertions = current_row[target_index] + ins_cost
            possible_substitutions = previous_row[target_index] + substitution_cost
            current_row.append(min(possible_insertions,
                                   possible_deletions,
                                   possible_substitutions))
        previous_row = current_row
    return previous_row.pop()
```

unit_test.py

```
#!/usr/bin/python3
""" unit_test.py: Unit testing for levenshtein string distance function
"""



import unittest
from string_converter import levenshtein



class UnitTest(unittest.TestCase):
    """UnitTest: Unit testing class for testing that levenshtein is working as expexted
    """
    def test_known_values(self):
        """test_known_values: Test values for vanilla string distance (levenshtein distance)
        """
        self.assertEqual(levenshtein('abc', 'abcd'), 1)
        self.assertEqual(levenshtein('abd', 'ab'), 1)
        self.assertEqual(levenshtein('abd', 'abc'), 1)
        self.assertEqual(levenshtein('kitten', 'sitting'), 3)
        self.assertEqual(levenshtein('hill', 'hello'), 2)

    def test_afaik_correct_values(self):
        """test_afaik_correct_values: This function does work for calculating the string
        distance but i'm not 100% sure when the weights are changed
        """
        self.assertEqual(levenshtein('abc', 'abcd', 3, 4, 5), 4)
        self.assertEqual(levenshtein('abd', 'ab', 3, 4, 5), 3)
        self.assertEqual(levenshtein('abd', 'abc', 3, 4, 5), 5)
        self.assertEqual(levenshtein('kitten', 'sitting', 3, 4, 5), 13)
        self.assertEqual(levenshtein('hill', 'hello', 3, 4, 5), 9)


if __name__ == '__main__':
    unittest.main()
```

Task 8 Part 2

main.py

```
#!/usr/bin/python3
""" main.py: File to hold the driver code to test hamiltonian for cycles
"""



from graph import Graph
from node import Node



def main():
    """main: Driver code to show working hamiltonian cycle finding
    """
    node0 = Node(0, [1, 3])
    node1 = Node(1, [0, 3, 4, 2])
    node2 = Node(2, [1, 4])
    node3 = Node(3, [0, 1, 4])
    node4 = Node(4, [1, 2, 3])
    graph = Graph([node0, node1, node2, node3, node4])
    for index, path in enumerate(graph.get_hamiltonian_cycles()):
        print('{0}: {1}'.format(index + 1, path))


main()
```

unit_test.py

```
#!/usr/bin/python3
""" unit_test.py: Testing for finding hamiltonian cycles
"""
```

```
import unittest
from graph import Graph
from node import Node
from weighted_graph import WeightedGraph
from weighted_node import WeightedNode


class UnitTest(unittest.TestCase):
    """UnitTest: There is no setup for this class
    """
    def test_known_no_hamiltonian(self):
        """test_known_no_hamiltonian: Test a graph which doesn't have a hamiltonian cycle
        """
        node0 = Node(0, [1, 3])
        node1 = Node(1, [0, 3, 4, 2])
        node2 = Node(2, [1, 4])
        node3 = Node(3, [0, 1])
        node4 = Node(4, [1, 2])
        graph = Graph([node0, node1, node2, node3, node4])
        self.assertEqual(graph.get_hamiltonian_cycles(), [])

    def test_known_hamiltonian_cycles(self):
        """test_known_hamiltonian_cycles: Test an unweighted undirected graph which has
        hamiltonian cycles
        """
        node0 = Node(0, [1, 3])
        node1 = Node(1, [0, 3, 4, 2])
        node2 = Node(2, [1, 4])
        node3 = Node(3, [0, 1, 4])
        node4 = Node(4, [1, 2, 3])
        graph = Graph([node0, node1, node2, node3, node4])
        self.assertEqual(graph.get_hamiltonian_cycles(), [[0, 3, 4, 2, 1, 0],
                                                          [0, 1, 2, 4, 3, 0],
                                                          [1, 2, 4, 3, 0, 1],
                                                          [1, 0, 3, 4, 2, 1],
                                                          [2, 4, 3, 0, 1, 2],
                                                          [2, 1, 0, 3, 4, 2],
                                                          [3, 4, 2, 1, 0, 3],
                                                          [3, 0, 1, 2, 4, 3],
                                                          [4, 3, 0, 1, 2, 4],
                                                          [4, 2, 1, 0, 3, 4]])

    def test_wiki_ham_cycle(self):
        """test_wiki_ham_cycle: This is the graph taken from wikipedia
        <https://en.wikipedia.org/wiki/Hamiltonian_path> and is also in this folder as a png called
        'HamCycle.png'
        This is the function which is making the unit test take a while becuase this
        the Hamiltonian path problem is NP-Complete
        """
        node0 = Node(0, [1, 4, 6])
        node1 = Node(1, [0, 2, 7])
        node2 = Node(2, [1, 3, 8])
        node3 = Node(3, [2, 4, 9])
        node4 = Node(4, [0, 3, 5])
        node5 = Node(5, [4, 10, 11])
        node6 = Node(6, [0, 11, 12])
        node7 = Node(7, [1, 12, 13])
        node8 = Node(8, [2, 13, 14])
        node9 = Node(9, [3, 10, 14])
        node10 = Node(10, [5, 9, 15])
        node11 = Node(11, [5, 6, 19])
        node12 = Node(12, [6, 7, 18])
        node13 = Node(13, [7, 8, 18])
        node14 = Node(14, [8, 9, 16])
        node15 = Node(15, [10, 16, 19])
        node16 = Node(16, [14, 15, 17])
        node17 = Node(17, [13, 16, 18])
        node18 = Node(18, [12, 17, 19])
        node19 = Node(19, [11, 15, 18])
        graph = Graph([node0, node1, node2, node3, node4,
                       node5, node6, node7, node8, node9,
                       node10, node11, node12, node13, node14,
                       node15, node16, node17, node18, node19])

        cycles = graph.get_hamiltonian_cycles()

        for cycle in cycles:
            self.assertEqual(len(cycle), 21)

        for cycle in cycles:
            self.assertEqual(len(set(cycle)), 20)

        self.assertTrue([0, 4, 3, 2, 1, 7, 13, 8, 14, 9, 10, 5, 11, 19, 15, 16, 17, 18, 12, 6, 0]
                        in cycles)

    def test_wiki_no_ham_cycle(self):
        """test_wiki_no_ham_cycle: This is also a graph taken from wikipedia
        <https://en.wikipedia.org/wiki/Hamiltonian_path> and there is no
        Hamiltonian cycles in this graph. This graph is also in included in this
        folder as a png called 'NoHamCycle.png'
        """
        node0 = Node(0, [7, 8, 10, 9])
        node1 = Node(1, [5, 6, 7, 10])
        node2 = Node(2, [6, 7, 8])
        node3 = Node(3, [5, 6, 8, 9])
        node4 = Node(4, [5, 9, 10])
        node5 = Node(5, [1, 3, 4])
        node6 = Node(6, [1, 3, 2])
        node7 = Node(7, [0, 1, 2])
        node8 = Node(8, [0, 2, 3])
        node9 = Node(9, [0, 3, 4])
        node10 = Node(10, [0, 1, 4])
        graph = Graph([node0, node1, node2, node3, node4,
                       node5, node6, node7, node8, node9, node10])
        self.assertEqual(graph.get_hamiltonian_cycles(), [])

    def test_no_weighted_hamiltonian(self):
        """test_known_no_hamiltonian_weighted: Test a weighted directed graph with no
        hamiltonian cycles
        """
        node0 = WeightedNode(0, [1, 3], {(0, 1): 4, (0, 3): 5})
        node1 = WeightedNode(1, [0, 3, 4, 2], {(1, 0): 2, (1, 3): 3, (1, 4): 6, (1, 2): 1})
        node2 = WeightedNode(2, [1, 4], {(2, 1): 2, (2, 4): 5})
        node3 = WeightedNode(3, [0, 1], {(3, 0): 2, (3, 1): 5})
        node4 = WeightedNode(4, [1, 2], {(4, 1): 3, (4, 2): 5})
        graph = WeightedGraph([node0, node1, node2, node3, node4])
        self.assertEqual(graph.get_hamiltonian_cycles(), [])

    def test_known_weighted_hamiltonian(self):
        """test_known_weighted_hamiltonian_cycles: Weighted directed graph which is known
        to have hamiltonian cycles
        """
        node0 = WeightedNode(0, [1, 3], {(0, 1): 4, (0, 3): 3})
        node1 = WeightedNode(1, [0, 3, 4, 2], {(1, 0): 3, (1, 3): 4, (1, 4): 4, (1, 2): 3})
        node2 = WeightedNode(2, [1, 4], {(2, 1): 1, (2, 4): 3})
        node3 = WeightedNode(3, [0, 1, 4], {(3, 0): 5, (3, 1): 3, (3, 4): 6})
        node4 = WeightedNode(4, [1, 2, 3], {(4, 1): 2, (4, 2): 6, (4, 3): 7})
        graph = WeightedGraph([node0, node1, node2, node3, node4])
        self.assertEqual(graph.get_hamiltonian_cycles(), [[0, 3, 4, 2, 1, 0],
                                                          [0, 1, 2, 4, 3, 0],
                                                          [1, 2, 4, 3, 0, 1],
                                                          [1, 0, 3, 4, 2, 1],
                                                          [2, 4, 3, 0, 1, 2],
                                                          [2, 1, 0, 3, 4, 2],
                                                          [3, 4, 2, 1, 0, 3],
                                                          [3, 0, 1, 2, 4, 3],
                                                          [4, 3, 0, 1, 2, 4],
                                                          [4, 2, 1, 0, 3, 4]])

    def test_get_shortest_hamiltonian(self):
        """test_get_shortest_hamiltonian: Test finding the short hamiltonian path
        or paths in a directed weighted graph
        """
        node0 = WeightedNode(0, [1, 3], {(0, 1): 4, (0, 3): 3})
        node1 = WeightedNode(1, [0, 3, 4, 2], {(1, 0): 3, (1, 3): 4, (1, 4): 4, (1, 2): 3})
        node2 = WeightedNode(2, [1, 4], {(2, 1): 1, (2, 4): 3})
        node3 = WeightedNode(3, [0, 1, 4], {(3, 0): 5, (3, 1): 3, (3, 4): 6})
        node4 = WeightedNode(4, [1, 2, 3], {(4, 1): 2, (4, 2): 6, (4, 3): 7})
        graph = WeightedGraph([node0, node1, node2, node3, node4])
        self.assertEqual(graph.get_shortest_hamiltonian_cycle(), [([0, 3, 4, 2, 1, 0], 19),
                                                                  ([1, 0, 3, 4, 2, 1], 19),
                                                                  ([2, 1, 0, 3, 4, 2], 19),
                                                                  ([3, 4, 2, 1, 0, 3], 19),
                                                                  ([4, 2, 1, 0, 3, 4], 19)])


if __name__ == '__main__':
    unittest.main()
```